공학박사 학위논문

Preventing Row-hammering and Improving Main Memory Performance by Exploiting Time Window Counters

타임 윈도우 카운터를 활용한 로우 해머링 방지 및 주기억장치 성능 향상

2020 년 8 월

서울대학교 융합과학기술대학원

융합과학부 지능형융합시스템전공

이 어 진

Preventing Row-hammering and Improving Main Memory Performance by Exploiting Time Window Counters

타임 윈도우 카운터를 활용한 로우 해머링 방지 및 주기억장치 성능 향상

지도 교수 안 정 호

이 논문을 공학박사 학위논문으로 제출함

2020 년 7 월

서울대학교 융합과학기술대학원

융합과학부 지능형융합시스템전공

이 어 진

이어진의 공학박사 학위논문을 인준함

2020 년 7 월

위 원 장 이 재 욱 (인)

부위원장 안 정 호 (인)

위 원 김 장 우 (인)

위 원 김 동 준 (인)

위 원 박 영 준 (인)

Abstract

Preventing Row-hammering and Improving Main Memory Performance by Exploiting Time Window Counters

Eojin Lee

Intelligence Systems

Department of Transdisciplinary Studies

The Graduate School

Seoul National University

Computer systems using DRAM are exposed to row-hammer (RH) attacks, which can flip data in a DRAM row without directly accessing a row but by frequently activating its adjacent ones. There have been a number of proposals to prevent RH, including both probabilistic and deterministic solutions. However, the probabilistic solutions provide protection with no capability to detect attacks and have a non-zero probability for missing protection. Otherwise, counter-based deterministic solutions either incur large area overhead or suffer from noticeable performance drop on adversarial memory access patterns.

To overcome these challenges, we propose a new counter-based RH prevention solution named Time Window Counter (TWiCe) based row refresh, which accurately detects potential RH attacks only using a small number of counters with a minimal performance impact. We first make a key observation that the number of rows that can cause RH is limited by the maximum values of row activation frequency and DRAM cell retention time. We calculate the maximum number of required counter entries per DRAM bank, with which TWiCe prevents RH with a strong deterministic guarantee. TWiCe incurs no performance overhead on normal DRAM operations and less than 0.7% area and energy overheads over contemporary DRAM devices. Our evaluation shows that TWiCe makes no more than 0.006% of additional DRAM row activations for adversarial memory access patterns, including RH attack scenarios.

To reduce the area and energy overhead further, we propose the threshold adjusted rank-level TWiCe. We first introduce pseudo-associative TWiCe (pa-TWiCe) that can search for hundreds of TWiCe table entries energy-efficiently. In addition, by exploiting pa-TWiCe structure, we propose rank-level TWiCe that reduces the number of required entries further by managing the table entries at a rank-level. We also adjust the thresholds of TWiCe to reduce the number of entries without the increase of false-positive detection on general workloads.

Finally, we propose extend TWiCe as a hot-page detector to improve main-memory performance. TWiCe table contains the row

addresses that have been frequently activated recently, and they are likely to be activated again due to temporal locality in memory accesses. We show how the hot-page detection in TWiCe can be combined with a DRAM page swap methodology to reduce the DRAM latency for the hot pages. Also, our evaluation shows that low-latency DRAM using TWiCe achieves up to 12.2% IPC improvement over a baseline DDR4 device for a multi-threaded workload.

Keywords : DRAM, Row-hammering, Deterministic protection, Reliability, Hot-page detection, Low-latency DRAM
Student Number : 2014-24902

Contents

List of Figures

List of Tables

Chapter 1

Introduction

DRAM, which is used as main memory in computer systems for decades, stores data by controlling the amount of charge per cell capacitor. Because a cell leaks charge over time, it should be refreshed periodically (once every refresh window (tREFW)) to retain data [3]. However, as process technology advances, individual DRAM cells become more susceptible to process variation, manufacturing imperfection, and influence from adjacent cells due to capacitive coupling. These reliability issues have been recognized as critical challenges to contemporary DRAM devices, and solutions such as sparing (groups of) DRAM cells and providing ECC capability within DRAM chips have been proposed and deployed [4]. In particular, row-hammering, a phenomenon that can flip data in

---

This Section is based on [1, 2]. –

adjacent (victim) rows and cause silent data corruption by repeatedly activating a specific (aggressor) DRAM row prior to its refresh window, has drawn public attention since 2014 [5].

Meanwhile, the capacity of a DRAM device has increased through process scaling and its bandwidth has improved by making its internal data-path wider and increasing the operating frequency of its inter-device I/O part [6]. Because the conventional focus of main-memory DRAM devices has been on higher storage density over cost, its access latency remains mostly unchanged and has improved (decreased) at a snail's pace. Also, existing commercial DRAM devices have symmetric access latency regardless of the topological location of DRAM cells [7].

In this dissertation, we propose a new counter-based row-hammering prevention solution named Time-Window Counters (TWiCe), and extend TWiCe as a hot DRAM row (page) detector to improve main-memory performance. TWiCe shows that strong, deterministic row-hammering protection and hot-page detection can be achieved by maintaining precise per-row ACT counts but only using a small number of counters.

1.1 Time Window Counter Based Row Refresh to Prevent Row-hammering

In order to mitigate or prevent the RH attacks, recent studies have proposed multiple protection techniques that refresh potentially

vulnerable rows earlier than its retention time [5, 8, 9, 10, 11]. PARA [5] provides probabilistic protection which can significantly reduce the probability of RH induced errors by also activating adjacent rows with a small probability for each DRAM row activation (ACTs). The probabilistic scheme is stateless and can be implemented with low complexity. Counter-based protection schemes, which deterministically refresh the adjacent rows when a row is activated more than a certain threshold, has also been proposed recently as an alternative protection approach. The counter-based schemes ensure that potential victim rows are always refreshed before the RH threshold is reached. The counter-based schemes also allow explicit detection of potential attacks, and enable a system to take action, such as removing/terminating or developing countermeasures for malware and penalizing malicious users responsible for the attack. The previous studies on counter-based protection schemes [9, 10, 12] pointed out that the performance overhead (the number of added ACTs) of the probabilistic schemes increases when stronger protection (lower error probability) is needed or the RH threshold decreases, whereas the counter-based schemes only issue additional ACTs when an attack is detected. Probabilistic and counter-based schemes provide different trade-offs between complexity and protection capabilities.

The main challenge in the counter-based protection schemes lies in reducing the cost of counters that track the number of ACTs.

Because maintaining a counter per row leads to prohibitive costs if they are kept in memory controllers (MCs), Counter-based Row Activation (CRA [8]) proposed to cache recently-used counters within MCs and store the remaining ones in main memory. The Counter-Based Tree (CBT [9, 10]) scheme proposes to track ACTs to a group of rows and dynamically adjust the ranges of rows each counter covers based on row activation frequency. Unfortunately, both CRA and CBT suffer from noticeable performance degradation on adversarial memory access patterns due to frequent counter cache misses and a flurry of refreshes on rows covered by a single counter, respectively.

To address this challenge, we propose a new counter-based RH prevention solution, named Time Window Counter (TWiCe) based row refresh. TWiCe guarantees to refresh victim rows before a RH threshold is reached only using a limited number of counters, which is orders of magnitude smaller than the total number of DRAM rows populated in the system. TWiCe is based on the key insight that the maximum number of DRAM ACTs over tREFW is bounded. This insight enables TWiCe to limit the total number of counters needed to monitor rows whose ACT counts may go over the protection threshold. TWiCe allocates a counter entry to a DRAM row only if the row is actually activated, and periodically invalidates (prunes) the entries if the corresponding rows are not frequently activated. Because tREFW is finite and row activation frequency in a DRAM

bank is limited by tRC (row cycle time), there is an upper bound on the number of ACT counter entries at any given time, leading to a low area overhead. We analytically derive the number of counters that are sufficient to monitor all potential aggressor rows. As TWiCe monitors each row individually, it guarantees a refresh before the number of ACTs exceeds a RH threshold.

We also explore the design space of where to place TWiCe, and carefully distribute the functionality of TWiCe across MCs, RCDs, and DRAM devices to minimize cost (e.g., area) and performance impact. We place the TWiCe counter entries (called TWiCe table) in RCDs because it is more cost-effective than placing them in MCs or DRAM devices. Placing the TWiCe table in a MC requires that the TWiCe table is large enough to accommodate the maximum number of DRAM banks that can be supported by the MC even when a system only contains much fewer DRAM banks, leading to a waste of resource in these typical cases. Placing a TWiCe table in each DRAM device is also wasteful because (around a dozen) devices in a DRAM rank operate in tandem and hence the TWiCe tables in all these DRAM devices would perform duplicated functionality.

Previously, both probabilistic and counter-based RH protection schemes are proposed to be implemented within MCs. However, this approach is difficult to realize in practice because modern DRAMs internally remap DRAM rows. The approach assumes that a MC knows which DRAM rows are physically adjacent, but it would be too

costly for a MC to store row remapping (replacing a row including faulty DRAM cells with a spare row) information of all DRAM devices it controls. To address this problem, we propose a new DRAM command, named ARR (Adjacent Row Refresh), to refresh the adjacent rows of an aggressor row because neither MC nor RCD (register clock driver) knows how DRAM rows are remapped. To avoid conflict between ARR and normal DRAM operations from MCs, we propose to provide a feedback path from RCD to MC, through which the RCD can send a negative acknowledgment signal when an ARR operation is underway in a DRAM bank.

Our analysis shows that there is no performance overhead on TWiCe table updates as it can be done concurrently with normal DRAM operations. The required TWiCe table size is just 3.11 KB per 1 GB bank, and energy overhead of table updates is less than 0.7% of DRAM activation/precharge energy. Also, our evaluation shows that TWiCe incurs no additional ACTs due to false positive detection on the evaluated multi-programmed and multi-threaded workloads and adds only up to 0.006% more ACTs on adversarial memory access patterns including RH attack scenarios; thus, the frequency of false positive detection is orders of magnitude lower than the previous schemes. These results show that precise counter-based RH protection is viable with low overhead.

1.2 Optimizing Time Window Counter

TWiCe requires a table with hundreds of entries per bank. Because the target row address of ACT can be stored in any entry of the table, it is straightforward to implement the table as a fully associative design, such as content-addressable memory (CAM). It is feasible because the minimal interval between two consecutive ACTs to a specific bank is dozens of nanoseconds, and the update is not in the critical path of DRAM access. However, it is energy-inefficient to searching CAM with hundreds of entries on every ACT.

To improve energy efficiency, we propose pseudo-associative TWiCe (pa-TWiCe) by leveraging a pseudo-associative cache design [13]. In pa-TWiCe, each DRAM row mapped to a preferred set, and the preferred set is first checked on ACT command. It is allowed only to use the entry of a non-preferred set when there is no available entry in the preferred set. Therefore, pa-TWiCe reduces energy consumption by reducing the number of table entries to be searched on ACT command without the eviction of entry due to thrashing.

Also, we optimize TWiCe by composing TWiCe as a rank level to reduce the area cost. The original TWiCe is designed based on the fact that the number of ACTs to a bank for a given time is limited by tRC, the ACT-to-ACT interval in a bank. For a device (rank), which is composed of multiple banks, TWiCe has to provide the table entries proportional to the number of banks. However, the number of ACTs to a rank is further limited by tRRD (Row to Row Delay) and tFAW

(Four Activated Window). Focusing on this property, we propose rank-level TWiCe and introduce the implementation of rank-level TWiCe by exploiting the structure of pa-TWiCe.

We further reduce the number of table entries by adjusting thresholds of TWiCe. We can adjust the thresholds of TWiCe within the extent that RH prevention is guaranteed. Especially, adjusting the threshold that determines the entries to be pruned can reduce the number of required entries. However, the thresholds should be carefully adjusted considering the increase of false-positive detections because the thresholds also affect the determination of the aggressor row. Therefore, we conduct experiments on how many rows are detected as aggressor rows on various workloads and reduce the number of TWiCe table entries by adjusting the thresholds as far as it does not increase the number of false-positive detection on general workloads.

1.3 Using Time Window Counters to Improve Main Memory Performance

The row-activation counts can also be used to identify frequently-accessed DRAM pages and to improve performance by allocating these pages to a low-latency region in asymmetric-latency DRAM designs. For example, CHARM [7] and TLDRAM [14] reduce access latency to a portion of a DRAM device by decreasing the number of DRAM cells that share sense amplifiers and hence accelerating data

acquisition speed. The system performance can be improved by allocating hot pages to this low-latency region of DRAM. Hot pages may be identified through offline profiling. However, this static approach is not effective for applications where hot pages change over time or can be affected by other applications on a system. We extend TWiCe to maximize performance improvement of low-latency DRAM architecture by dynamically detecting hot pages and migrating them to a fast region of DRAM at runtime.

For the runtime migration approach to be effective, we need a low-overhead method to swap data between DRAM rows and to translate DRAM addresses. Because a DRAM row (typically 8KB) consists of dozens of cache lines (around 64B), relying on a CPU to move data in DRAM can take more than a microsecond, negating the performance benefit of the lower DRAM access time. We leverage previous proposals for high-throughput data transfers within a DRAM device such as RowClone [15], LISA [16], and DAS-DRAM [17] for fast page swapping, and introduce an address translation table. Through a detailed timing analysis, we show that the proposed swap methodology and the address translation table management method are feasible without much overhead.

Our performance evaluation shows that low-latency DRAM using TWiCe with a hot-page detection threshold value of 16 improves IPC by 5.6% and 12.2% for multi-programmed workloads using SPEC CPU2006 benchmarks and RADIX multi-threaded workload,

respectively. Overall, the results show that TWiCe can be used to intelligently manage data placement in the asymmetric DRAM architecture to improve performance.

1.4 Outline

The organization of this dissertation is as follows.

In Chapter 2, we describe the organization and operation of DRAM device and main memory subsystem. Also, we introduce row-hammering (RH) phenomenon and the previous RH prevention solutions.

Chapter 3 describes the proposed RH prevention solution, which uses Time Window Counters (TWiCe), and Chapter 4 shows the optimization techniques of TWiCe to reduce implementation cost. In Chapter 5, we introduce augmenting TWiCe for hot-page detection. Finally, we present conclusions and future works in Chapter 6.

Chapter 2

Background of DRAM and Row-hammering.

A modern server typically manages trillions of DRAM bits for main memory owing to technology scaling [18, 19, 20]. This enables unprecedented benefits to applications with diverse performance and capacity requirements. At the same time, however, the finer fabrication technology entails a number of challenges on organizing and operating a main memory system because the massive number of DRAM cells should be hierarchically structured for high area efficiency (to lower cost) and more cells become faulty (either permanently or intermittently) due to process variation and manufacturing imperfection [4, 21, 22]. This chapter reviews the details of the main memory organization and operations, which must

This Section is based on [1, 2]. –

be considered when designing a solution for row-hammering (RH).

2.1 DRAM Device Organization

A server includes dozens to hundreds of DRAM devices. A DRAM device consists of billions of cells, each comprised of an access transistor and a capacitor [6, 23]; the amount of charge in the capacitor represents data: either zero or one (see Figure 2.1). Cells in a DRAM device are grouped into multiple (typically around 16 these days) banks. A bank is further divided into thousands of mats structured in two dimensions. A group of mats that share global wordlines (WLs) and hence operate together is called a subarray. Within a mat, cells are again organized in two dimensions; cells that are aligned in a row share a local WL and the ones aligned in a column share a bitline (BL) to increase area efficiency.

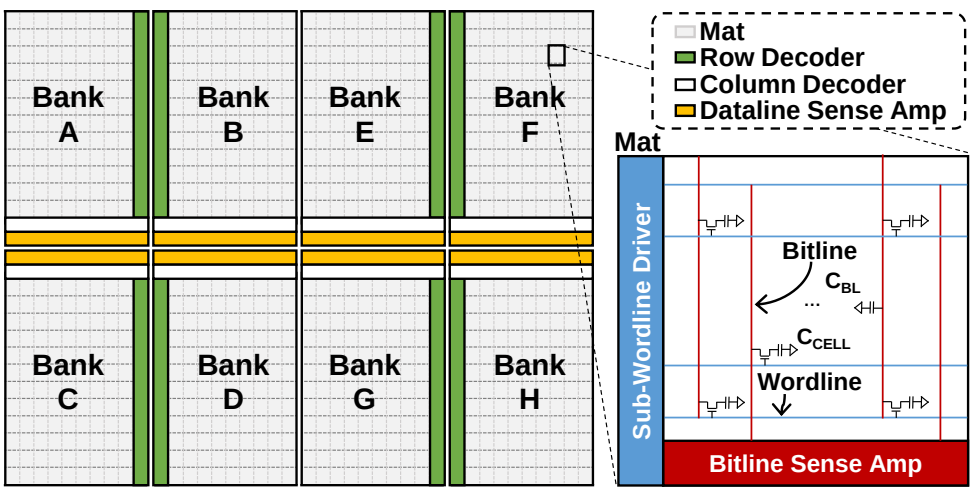


Figure 2.1. The organization of a modern DRAM device.

A DRAM device periodically refreshes each cell within retention

time called *tREFW* (refresh window). Because a cell discharges (leaks) slowly but steadily, data is lost unless DRAM periodically performs a refresh operation to restore the charge to a cell capacitor. As the number of rows per bank increases continuously to provide higher DRAM capacity, a modern DRAM bank refreshes not a single row but a set of rows per auto-refresh operation. The number of rows refreshed per auto-refresh increases over time; so does its duration called *tRFC* (refresh command time) performing an auto-refresh operation. The interval between two auto-refresh operations, called *tREFI* (refresh interval), is $\frac{tREFW}{\#\ of\ row\ sets}$.

2.2 Sparing DRAM Rows to Combat Reliability Challenges

Wire pitch gets finer, and storage cells become smaller as fabrication technology advances. It exacerbates the impact of process variation and manufacturing imperfection, increasing the probability of functional and timing failures of storage devices including DRAM devices [4].

Therefore, faulty DRAM cells are corrected using various techniques. Replacing a row or a column of a DRAM bank with faulty cells with another fault-free row or column (row/column sparing) is a conventional method, which has been employed in commodity DRAM devices [21]. Another method which is gaining momentum in fixing faulty DRAM cells is in-DRAM ECC [4], which corrects up to

a few errors in a block of bits (called codeword) through error correcting codes using parity bits in addition to data bits. In this paper, we focus on more traditional row sparing method, which also influences main memory DRAM organization and operations.

Each DRAM bank is equipped with spare rows and columns that can replace faulty rows, columns, and cells. These spare rows/columns are set up as follows. During the test phase of DRAM device fabrication, test equipment identifies the locations of faulty cells. A repair algorithm calculates and assigns target spare rows and columns for the faulty cells, columns, and rows to efficiently leverage these spares. The information pairing the addresses of a faulty row/column and the corresponding target one (called remapping hereafter) is stored in a one-time programmable memory, such as electrical fuses within a DRAM device [21].

The locations of malfunctioning DRAM cells are different for individual DRAM devices; hence it is reasonable to place the cell repair functionality within DRAM devices. An important implication of this row sparing is that due to this remapping, the rows whose index numbers differ by one in a DRAM bank is not necessarily physically adjacent within a DRAM device.

2.3 Main Memory Subsystem Organization and Operation

As depicted in Figure 2.2, a conventional main memory system consists of a group of memory controllers (MCs). One MC handles one or a few memory channels. A channel is connected to a small number (typically fewer than four) of dual-inline memory modules (DIMMs). Each module consists of a few ranks, each having several DRAM devices. All DRAM devices within a rank operate in tandem.

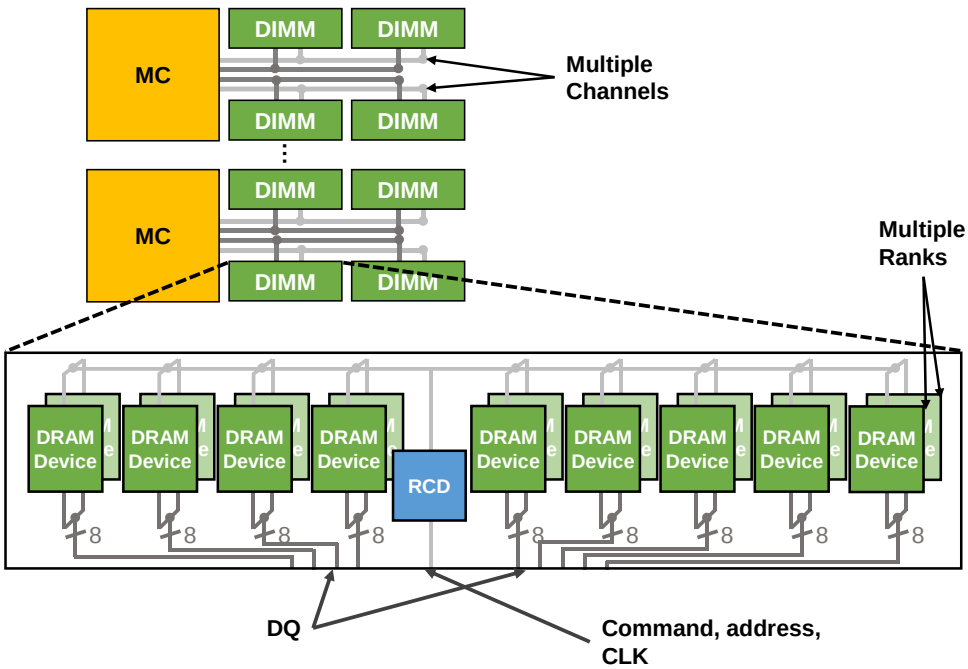


Figure 2.2. The organization of a conventional main memory system. Each memory controller (MC) can populate multiple DIMMs, and each DIMM consists of one or a few ranks. Each rank has several DRAM devices which operate in tandem.

Modern servers have dozens of cores per CPU socket and multiple MCs to provide enough main memory bandwidth to the cores [18, 19]. Also, the emergence of virtual machines and containers demands large main memory capacity per CPU; and hence typically

multiple DIMMs are connected to a memory channel. Therefore, the command and address (CA) signals from a MC through one of its memory channels have to be broadcasted to dozens of DRAM devices, imposing a huge channel load in driving these signals.

To mitigate this signal integrity problem, the CA signals and optionally data signals from a MC are buffered within a modern DIMM but outside of DRAM devices of the module. The separate buffer device is called a register clock driver (RCD [24]). A registered DIMM (RDIMM [25]) only repeats CA signals, reducing the load from a MC, with additional latency *tPDM* (propagation delay). A load-reduced DIMM (LRDIMM [26]) repeats both CA and data signals; the data signals can be repeated in the same RCD chip (DDR3) or in the separate devices (called data buffers in DDR4)

A MC receives an access (read or write) request with an accompanying address, translates the address into a tuple of (memory channel, rank, bank, row, column), and generates one or more DRAM commands to serve the request. The number of DRAM commands per request and the timing of each command depend on the internal states of a MC (including other requests stored in the request queue) and various timing constraints. Because conventional memory interfaces, such as DDR [27], GDDR [28], and LPDDR [29], adopt a primary-secondary (master-slave) communication model, only a MC generates commands within a memory channel and it knows when the DRAM devices it controls reply data, owing to the

synchronous nature of the interface.

If the target bank of a request does not have an active row (BLs being precharged to $\frac{VDD}{2}$), an activate command (ACT) is issued and a high voltage level is applied to the global WL (whose target row is specified by the physical address of the request), enabling BL sense amplifiers (BLSAs) to detect and latch the data stored in the target row within *tRCD* (row access to column access delay). The data of the target column latched in the BLSAs are transferred to the I/O pads of the corresponding DRAM device through the global dataline, which takes *tCL* after a read command (RD) is issued (the data transfer direction is flipped for a write command (WR)). In the course of an activation process, the voltage level of the selected cells is first changed close to $\frac{VDD}{2}$ as they share charges with BLs whose capacitance is much larger than that of a DRAM cell, but is then restored to either VDD or ground after *tRAS* because BLSAs amplify the voltage level.

If the target bank has an active row which is the same as the target row, ACT is omitted, and hence the data can be accessed faster. If the currently active row of the target bank is different from the target row, the row must be deactivated first; the voltage level of BLs must be set to $\frac{VDD}{2}$ by sending a precharge command (PRE), which takes *tRP* (row precharge time) after which the (next) target row is ready to be activated.

Each DRAM bank processes these command sequences

independently. However, the frequency of issuing ACTs to a DRAM device is limited by *tRRD* (minimum time between any two ACTs) and *tFAW* (minimum interval between a group of four ACTs). Within a DRAM bank, *tRC* (minimal time between two ACTs to the same bank) limits the frequency of row activation.

The row address (index) from a MC may target one with faulty DRAM cells. A comparator within a DRAM device identifies this address and replaces it with a spare row before the row decoder decodes the incoming row address. This remapping breaks the tie between logical (index being offset by one) and physical adjacency (and hence interfering with each other due to capacitive coupling) of DRAM rows.

2.4 Row-hammering (RH)

Row-hammering (RH) is a DRAM reliability challenge, which has gained significant public attention due to its security implications. RH is an attack that exploits the phenomenon that repeated activations to a specific (aggressor) DRAM row cause bit flips in its adjacent (victim) rows before the victim rows reach their retention time limits (*tREFW*), which is publicly reported by Kim et al. in 2014 [5]. RH effectively reduces DRAM cell retention time depending on access patterns, making data preservation difficult. Park et al. [30] explained the root cause of this RH. They found out that during a row activation and precharge operation, a portion of electrons in the chosen WL

flows into the cells of the adjacent rows with a low probability. Repeated activation and precharge operations make the number of electrons passed surpass a certain threshold, causing the data to be flipped.

Then, studies have shown that RH can be exploited to compromise real-world systems without software vulnerability [31, 32, 33]. Flip Feng Shui [32] accesses a co-hosted virtual machine in an unauthorized way through a combined use of memory deduplication (identifying an RSA public key) and RH (flipping the key). Drammer [33] takes control of a mobile device running Android by performing RH attacks on specific parts of the device's memory. These attacks highlight the importance of providing adequate solutions to RH.

In order to avoid errors from row-hammering, a DRAM row needs to be refreshed before adjacent rows are activated too many times. Similar to the DRAM refresh window, we expect a DRAM vendor to provide a new parameter, named a row-hammer (RH) threshold, which specifies the maximum number of ACTs on the adjacent rows within an interval of *tREFW* before a row needs to be refreshed. The DRAM vendor ensures that a row will not have an error before its RH threshold is reached similar to ensuring that the DRAM retention time is longer than the refresh window. While exceeding the RH threshold does not mean there will be an RH error, there is no guarantee on reliability once the threshold is exceeded.

Therefore, the job of a system designer is to ensure that each row is refreshed before it exceeds the RH threshold, which is expected to decrease going forward with further technology scaling [34].

2.5 Previous RH Prevention Solutions

Previous architectural solutions against the RH attack can be categorized into two groups: counter-based and probabilistic RH protection schemes. As the likelihood of RH increases after a large number of ACTs are sent to a DRAM row, a naive counter-based solution would record the number of ACTs for each row and refresh a victim row once the ACT count exceeds the RH threshold. However, this scheme requires a counter per DRAM row, leading to prohibitive costs especially if the counters are kept in MCs because a MC covers more than millions of DRAM rows. Counter-based Row Activation (CRA [8]) counts ACTs for all DRAM rows but stores only the ACT counts for frequently activated rows in caches located at MCs and all remaining counters in DRAM.

CBT [9, 10] reduces the number of counters by having each counter track ACTs to a group of rows. The group size is determined dynamically based on the ACT frequency to the group; a counter covers a small number of hot (frequently activated) rows or a large number of cold rows. The counters in CBT are organized as a non-uniform binary tree, where each counter at the same tree level (distance from the root) covers the same number of DRAM rows.

Initially, CBT uses only one counter to track the number of ACTs for all DRAM rows together. Once the count exceeds a threshold, two child counters at the next tree level are used, each counting the ACTs to the half of the DRAM rows covered by the parent. The children are initialized to the value of the parent. CBT repeats this process until all counters are used up and resets the tree every *tREFW*.

To reduce counter overhead, another counter-based approach that uses system performance counters [35, 36] has been proposed. It monitors the last level cache (LLC) misses and regards unusually frequent LLC misses as a row-hammer attack. However, it requires an action for preventing row-hammering whenever there are frequent LLC misses, resulting in substantial performance overhead.

In addition to the counter-based protection schemes, previous studies also proposed probabilistic protection schemes. For example, PARA [5] activates adjacent DRAM rows with a low probability whenever a row is precharged. By adjusting the probability, PARA can choose a trade-off point between the level of protection against RH attacks and performance and energy overhead. PRoHIT [11] extends PARA with a history table to activate the adjacent rows of more frequently activated rows with a higher probability.

2.6 Limitations of the Previous RH Solutions

Even if the previous proposals advanced the state-of-the-art against the RH attacks compared to the naive counter-based scheme,

they suffer from the following shortcomings. Counter-based approaches can provide strong protection with no false negative by identifying all rows whose ACT counts exceed a threshold value, but they can suffer from system performance degradation due to superfluous DRAM operations on adversarial memory access patterns.

In the case of CRA, counter-cache misses amplify main memory accesses. Similar to other caches, the counter cache within a MC is not effective if memory access patterns do not exhibit enough locality (being adversarial to the cache). Especially in random access workloads, the number of ACTs is nearly doubled, which can seriously degrade the system performance.

CBT may generate bursts of DRAM refreshes due to false positives depending on memory access patterns. Because one counter often covers multiple DRAM rows, all rows within a group, including ones that are not heavily activated, need to be refreshed together when the total number of ACTs for the group (as many as half the number of rows in a bank) exceeds the threshold. This flurry of refreshes incurs a spike in memory access latency, which hurts latency-critical workloads [37, 38], degrading their overall system performance. Moreover, when a parent counter is split into children, ACTs are counted twice because the two child counters are initialized with the value of one parent counter.

PARA and PRoHIT can significantly reduce the probability of an

RH-induced error with low performance and energy overhead. Yet, the protection is probabilistic in nature; while the probability is quite small, there is a non-zero probability that a victim row is not refreshed after reaching its RH threshold. The previous studies on counter-based protection schemes [9, 12] point out that the performance overhead (# of added ACTs) of the probabilistic schemes increases when stronger protection (a lower error probability) is needed or if the RH threshold decreases. The counter-based scheme can be a more cost-effective solution if a system designer wants to ensure that the RH threshold is never exceeded similar to the way that today' s refresh mechanisms deterministically refresh a row within the refresh window. PARA and PRoHIT are also oblivious to the RH attack; while they reduce the probability of RH errors, they cannot pinpoint when and where an attack attempt is made. By contrast, the counter-based schemes explicitly detect an RH attack and enables a system to take action such as removing/terminating or developing countermeasures for malware, and penalizing malicious users responsible for the attack. For probabilistic schemes, attackers can easily avoid refreshes for a victim row if they can predict the output of a random number generator. In that sense, it is important to ensure that the random numbers are unpredictable, possibly using true random number generators (RNGs) rather than pseudo RNGs.

All previous techniques are proposed to be implemented within

MCs, but this is not necessarily ideal for combatting the RH attack due to the following reasons. They assume that MCs know physical adjacency among rows, possibly by obtaining the mapping information between logical and physical rows from DRAM devices. However, due to inevitable remapping of DRAM rows as described in Section 2.2, it is costly to know the remapping information. For example, the single-cell failure rate (SCF) of a DRAM device is projected to be around or surpass 10-5 in sub-20nm DRAM process technologies [4]. In this case, if one MC populates DRAM capacity of 64 GB, it should retain more than 5 million remapping information to know the physical adjacency of the entire rows it controls. It is impractical or highly costly to have all of this information in each MC.

	CRA [8]	CBT [9, 10]	PARA [5]	TWiCe
Primary location	MC	MC	MC	RCD
Performance drop on typical memory access patterns	Small	Smaller	Small	No
Performance drop on adversarial memory access patterns	High	High	Small	Smaller
Possibility of RH attack detection	Yes	Yes	No	Yes

Table 2.1. Comparing TWiCe with previous row-hammer prevention/mitigation solutions.

Moreover, because MCs control a varying number of DRAM devices and there is a huge variation in the DRAM capacity, previous proposals that are implemented within MCs must support the worst case (e.g., the maximum number of DRAM rows that one MC may control). For the counter-based approaches, this means that the counters must be provisioned assuming the maximum possible number of rows. Because the actual main memory capacity can be much lower than the maximum depending on workloads, this often leads to a waste of resources. Table 2.1 summarizes the properties and limitations of the existing solutions and proposed solution, TWiCe, which is described from the next chapter.

Chapter 3

TWiCe: Time Window Counter based RH Prevention

In order to prevent RH precisely with low cost, we propose a new counter-based RH mitigation solution named TWiCe (Time Window Counters). Based on the insight that the number of DRAM ACTs over *tREFW* is bounded, TWiCe prevents RH with a small number of counters.

3.1 TWiCe: Time Window Counter

Naively dedicating a counter per DRAM row would be prohibitively expensive because the number of necessary counter entries is proportional to ever-growing memory capacity. For example, if the main memory capacity of a system is 1 TB and a

This Section is based on [1, 2]. – © 2019 ACM, and IEEE 2018.
Reprinted, with permissions from ISCA '19, and CAL '18.

DRAM page size is 8 KB, more than 100M counters are needed. The number of counter entries can be reduced in theory as not all DRAM rows can be simultaneously susceptible to the RH attack. A row is refreshed every *tREFW*. This resets the number of electrons that could be piled up due to the RH attack. Therefore, if the RH attack on a row is spread over a duration spanning multiple *tREFW*, only the number of ACTs a row experiences within *tREFW* from its physically adjacent rows matters. If this number surpasses the RH threshold (N_{th}), data in the corresponding row may be flipped.

The maximum frequency of row ACTs is limited. On a DRAM bank, the minimum interval between any two ACTs is *tRC* (bank cycle time), limiting the maximum number of ACTs within the retention time (*tREFW*) of a row to $\frac{tREFW}{tRC}$. Assuming that a row activation affects two adjacent (victim) rows, at most $2 \times \frac{tREFW}{tRC \times N_{th}}$ rows experience the RH attack within *tREFW*. Applying typical values on modern DRAM chips (*tRC* = 45.32 ns, *tREFW* = 64 ms) and N_{th} value reported in [26] (N_{th} = 139K), only up to 20 rows can be exposed to the RH attack from a bank in the duration of *tREFW*. Therefore, we can decrease the number of counter entries by detecting the rows that have the potential to be RH aggressors and only counting the ACTs to those rows, which is a key idea of TWiCe.

TWiCe guarantees protection against the RH attack by precisely counting ACTs for individual DRAM rows but has low overhead because the counts are kept only for frequently activated DRAM rows.

The number of necessary counters can be bounded because the DRAM interface limits the maximum frequency of row ACTs, and the ACT count only needs to be tracked within a refresh window (*tREFW*). We further reduce the number of counters in TWiCe by periodically removing (pruning) the counts for the rows that are activated infrequently. We refer to this time window period as a pruning interval (PI). We can mathematically show that the ACT counts for such infrequently activated rows are unnecessary for an RH protection guarantee and that TWiCe guarantees to prevent RH attacks. The parameters and example values for TWiCe are summarized in Table 3.1; we illustrate TWiCe with DRAM whose *tREFW*, *tREFI*, and *tRC* are 64 ms, 7.8125 μs, and 45.32 ns, respectively.

Term	Definition	Typical value
tREFW	refresh window	64 ms
tREFI	refresh interval	7.8125 μs
tRFC	refresh command time	350 ns
tRC	ACT to ACT interval	45.32 ns
th_{RH}	RH detection threshold	32,768
th_{PI}	pruning interval threshold	4
max_{act}	max # of ACTs during PI	164
max_{life}	max $life$ of a row in PI	8,192

Table 3.1. Definition and typical values of TWiCe.

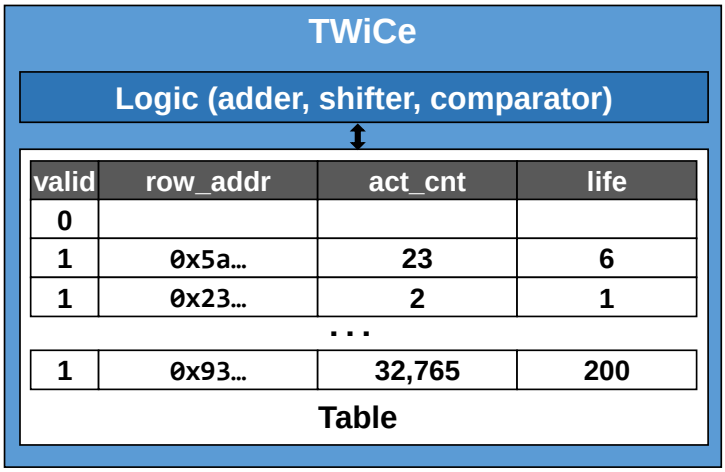


Figure 3.1. The organization of TWiCe. Each table entry holds $valid\_bit$, row_addr, act_cnt, and $life$. An entry is inserted when a new row is activated and invalidated when pruned or refreshed after act_cnt reaches th_{RH}.

TWiCe consists of a counter table and counter logic (Figure 3.1). Each counter table entry contains row_addr, act_cnt, $valid\_bit$, and $life$. act_cnt records the number of ACTs to the target row_addr. $valid\_bit$ indicates whether the entry is valid. $life$ indicates the number of consecutive pruning intervals (PIs), for which the entry stays valid in the table.

We define two threshold values, one to identify RH (th_{RH}) and the other to detect aggressor candidates (th_{PI}). Similar to other counter-based approaches, TWiCe refreshes adjacent rows if act_cnt exceeds th_{RH}. th_{PI} determines whether an entry should be kept as an aggressor candidate after each PI. We set the PI to match the auto-refresh interval (*tREFI*) to hide the latency of checking the

table entries by performing the operation in parallel with an auto-refresh. As each row is refreshed once every refresh window (*tREFW*), the number of ACTs to a row must exceed th_{RH} within *tREFW* for a successful RH attack. Thus, the average number of ACTs to an aggressor row over a refresh interval (*tREFI*) must exceed $\frac{th_{RH}}{tREFW/tREFI}$. We set th_{PI} to be this value. For the DRAM parameters that we use, *tREFW* = 64 ms and *tREFI* = 7.8125 μs, th_{PI} is 4 and the maximum number of pruning intervals over a refresh window (max_{life}) is 8,192.

TWiCe operates as follows (see Figure 3.2). 1) TWiCe receives a DRAM command and address pair. 2) For each DRAM ACT, TWiCe allocates an entry in the counter table if the entry for the row does not already exist, and increments the counter (act_cnt) by one. 3) If act_cnt reaches th_{RH}, TWiCe refreshes the adjacent rows of the entry and deallocates the entry. 4) After each pruning interval (PI = *tREFI*), each entry in the TWiCe table is checked and removed if ($act\_cnt < th_{PI} \times life$). In other words, a row is considered to be an aggressor candidate only if the average number of ACTs over *tREFI* is equal to or greater than th_{PI}. This step enables the counter table size to be bounded. For the remaining entries, $life$ is incremented by one.

3.2 Proof of RH Prevention

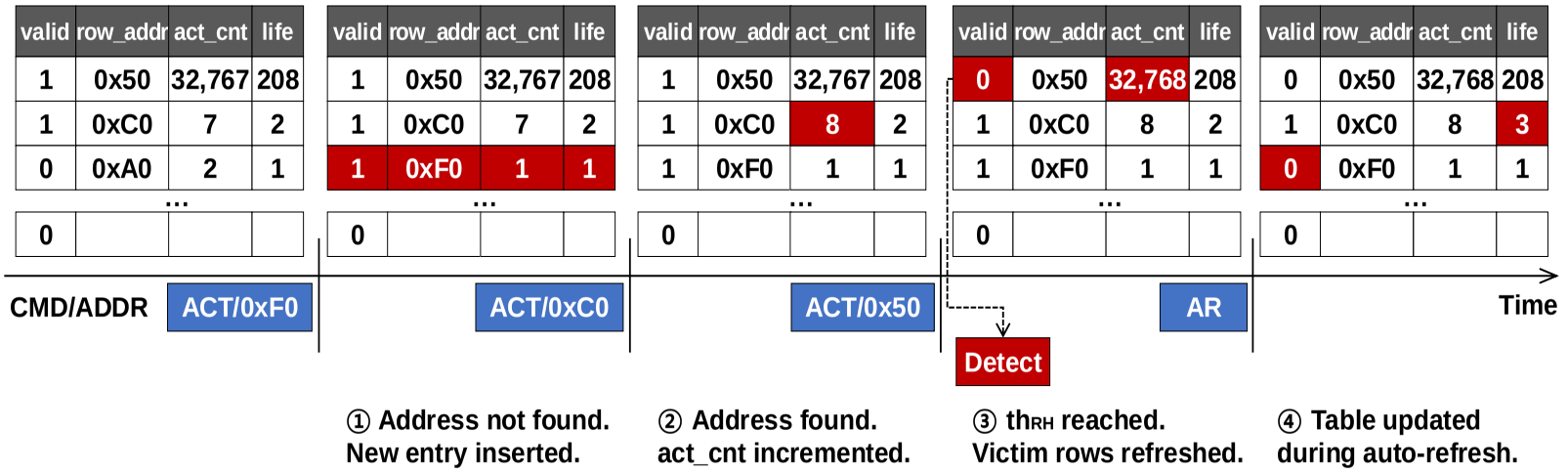


Figure 3.2. TWiCe operation example. The DRAM command/address and changes in TWiCe are colored blue and red, respectively. ① When the target address of ACT is not found, a new entry is inserted. ② When found, act_cnt is incremented by 1. ③ If act_cnt reaches th_{RH}, the victim rows are refreshed and the entry is invalidated. ④ During an auto-refresh, the table is updated; the aggressor candidates' $life$ is increased by 1, while others are pruned.

Here, we show that the number of ACTs to each row over a refresh window cannot exceed the RH threshold without being detected by TWiCe. Let us first consider the maximum number of ACTs to a row over tREFW when the row is not tracked by the TWiCe table ($count_{not-tracked}$). Because TWiCe keeps a row in its counter table if $act\_cnt \geq th_{PI} \times life$, $count_{not-tracked}$ must be less than $th_{PI} \times life$. Given the maximum value of life over the refresh window is $tREFW/tREFI$ and th_{PI} is $\frac{th_{RH}}{tREFW/tREFI}$, $count_{not-tracked}$ can be expressed as:

$$count_{not-tracked} < th_{PI} \times \frac{tREFW}{tREFI} = th_{RH}$$

In other words, if a row is activated th_{RH} times or more within a refresh window, it will be in the counter table.

If a row is in the counter table, its ACT count while being considered as an aggressor candidate ($count_{tracked}$) is less than th_{RH} if no RH attack is detected. The activations to this row, while it was not considered as an aggressor candidate, may not be included in the counter table, yet this value is bounded by $count_{not-tracked}$, which is less than th_{RH}. As explained above, both counttracked and the invalidated counts $count_{not-tracked}$ should be less than th_{RH}. Therefore, the maximum number of ACTs to a row over *tREFW* without being detected as an aggressor ($count_{combined}$) is

$$count_{combined} = count_{not-tracked} + count_{tracked} < 2 \times th_{RH}$$

According to a previous study [5], a row needs to experience 139K or more ACTs on its neighbor rows within *tREFW* to have a bit flip (N_{th}). Considering that a row has two adjacent rows in general (double-side RH), the actual threshold to detect an aggressor is its half, 69K. In order to ensure that $count_{combined}$ does not exceed this threshold, 69K, th_{RH} should be less than half of 69K (or one-fourth of N_{th}. In this study, we set th_{RH} to be 32,768.

3.3 Counter Table Size

In TWiCe, we assume that there is a counter table per DRAM bank. To calculate the required table size (the number of counter entries), we define a new term max_{act}, the maximum number of ACTs in a DRAM bank during *tREFI*. Because the ACT-to-ACT interval in a bank is *tRC* and rows cannot be activated during *tRFC*, max_{act} is $(tREFI - tRFC)/tRC$. With *tREFI* of 7.8125 μs and *tRC* of 45.32 ns, max_{act} is 164. DRAM devices with fewer rows per bank lead to smaller *tRFC* and higher max_{act}. Yet, because *tREFI* $\gg$ *tRFC*, max_{act} only changes slightly.

The table size should be set based on the worst case when the table has the largest number of valid entries (aggressor candidates). If there are not enough TWiCe table entries to handle all the aggressor candidates, overflows cause entry evictions. In this case,

information lost by eviction makes it hard to prevent row-hammering through TWiCe. Although refreshing adjacent rows of evicted row entry can solve this problem, it will enable the system performance degradation attack that uses adversarial memory access patterns, which evicts TWiCe table entries frequently.

The valid entries fall into two categories: (1) entries newly inserted in the current PI, and (2) entries identified as aggressor candidates in the previous PIs. The number of new entries is bounded by max_{act}. The number of surviving entries is maximized when the counter entries with the smallest $life$ survive the most. For example, consider the entries whose $life$ is 2. Because $life$ of these entries in the previous PI is 1, the maximum number of entries with $life$ = 2 is $\frac{max_{act}}{1\times th_{PI}}$. This happens when the maximum number of ACTs (max_{act}) are equally distributed across $\frac{max_{act}}{1\times th_{PI}}$ distinct rows in the previous PI. New entries with fewer than th_{PI} ACTs are invalidated at the end of the PI. Similarly, the maximum number of entries whose $life$ is n can be calculated as $\frac{max_{act}}{(n-1)\times th_{PI}}$. Thus, the total number of counter entries can be bounded by $max_{act} \times (1 + \sum_{n=1}^{max_{life}} \frac{1}{n\times th_{PI}})$. Moreover, the number of entries must be an integer, so $\{max_{act}\%\big((n-1)\times th_{PI}\big)\}$ of ACTs, which are left after filling $((n-1)\times th_{PI})$ counters at $life$ of n, can be used for entries with $life$ of $n+1$. For example, with th_{PI} of 4 and max_{act} of 164 in Table 3.1, the maximum number of entries whose life is 3 and 4 is 20 and 13, respectively, according to the formula above. Also, four (=164−20×8) and eight (=164−13×12)

ACTs remain in the corresponding PIs, respectively. In this case, these 12 remain ACTs can be used for saving one more valid entry whose life is 4.

The maximum number of entries per TWiCe table is 553 by the formula shown above, while the total number of rows per bank is 131,072 for the parameters in Table 3.1. Therefore, the required table size is reduced by more than two orders of magnitude compared to the number of DRAM rows in a bank, which is comparable to other counter-based approaches.

3.4 Architecting TWiCe

TWiCe can be implemented in multiple ways by placing its counter table and RH detection logic in a MC, a DRAM device, or an RCD. In this section, we discuss this design space and describe how we modify MC, RCD, and DRAM devices to support TWiCe in main memory systems. This section also introduces a new Adjacent Row Refresh (ARR) command that is necessary to deal with row remapping within DRAM devices.

3.4.1 Location of TWiCe Table

TWiCe needs one table per DRAM bank. A certain class of systems, such as mobile devices, has a fixed number of DRAM banks whereas another class of systems, such as servers, could have a varying

number of banks in their life time. As a result, if we locate a TWiCe table in a MC, the number of TWiCe tables must be large enough to

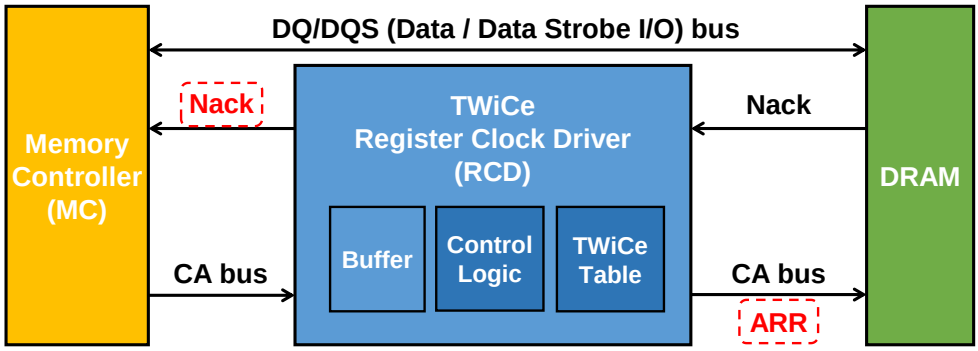


Figure 3.3. The microarchitecture of TWiCe. TWiCe table is implemented in a register clock driver (RCD). A path from an RCD to its master memory controller (MC) is modified to send negative acknowledgment (nack) signals. A new command called adjacent row refresh (ARR) is sent to DRAM devices from RCD through the repeated command and address (CA) bus when the row address specified in ACT is identified as an RH aggressor.

accommodate the largest number of DRAM banks the MC might support, not the actual number of DRAM banks in a system. For example, a MC, which could populate a maximum of four 2-rank DIMMs with 16 banks per rank, must be designed with TWiCe tables that support up to 128 banks. If this MC controls only one 1-rank DIMM with 16 banks, TWiCe tables for the 112 banks are unused and hence wasted.

Implementing the TWiCe table within each DRAM device is also wasteful when a DRAM rank consists of multiple DRAM devices. All DRAM devices within a memory rank operate in tandem, hence

making each DRAM device count the number of ACTs from the MC would be a duplication of effort. Placing the TWiCe counters in an RCD would provide a per-DIMM protection, avoiding table size over-provisioning, and count the number of ACTs at a per-bank level, eliminating redundant information. Therefore, in this paper, we investigate placing the TWiCe table in an RCD (see Figure 3.3).

3.4.2 Augmenting DRAM Interface with a New Adjacent Row Refresh (ARR) Command

As we explained in Section 2.2, row remapping occurs within DRAM devices, but neither MC nor RCD knows this DRAM row remapping information or can efficiently hold all the information internally. Therefore, an RCD should not compute adjacent rows and send the computed addresses explicitly to DRAM devices.

Instead, the RCD should just send a command to DRAM devices notifying that the row of a bank which was just activated are recognized as an RH aggressor row. Hence, we add a new DRAM command ARR (Adjacent Row Refresh) which asks the DRAM devices to refresh the physically adjacent rows of the row just being activated (through up to two pairs of ACTs and PREs within the devices). When TWiCe detects an RH aggressor row and the RCD equipped with TWiCe receives a precharge command (PRE) to the aggressor row, the RCD sends ARR to the DRAM devices instead of PRE and waits for $2 \times tRC + tRP$ to allow the DRAM to refresh the

(up to two) physically adjacent rows and return the bank to a precharged state. DRAM devices receiving an ARR command calculate the physical addresses of the adjacent rows (considering the row remapping) during the precharge operation of the aggressor row and then refresh them.[①]

We also propose to provide a feedback path from an RCD to a MC for sending negative acknowledgment information (see Figure 3.3). Updating a TWiCe table is asynchronous to normal DRAM operations because the update happens when the corresponding bank performs an auto-refresh operation, not accepting any normal DRAM command, such as RD, WR, ACT, and PRE. Therefore, MCs do not need to know about a TWiCe table update as long as the update can be performed within *tRFC* (which is analyzed in Section 3.5).

By contrast, because an RCD with TWiCe sends ARR right after a row being recognized as an RH aggressor is precharged, one of normal DRAM operations from a MC to the RCD might head to the DRAM bank that is still performing ARR, leading to a conflict. Conventional DRAM interfaces assume that a MC is a master, a sole device which generates commands and expects the other devices (here DRAM devices) to process the commands without any internal delay mechanism. Fortunately, ARR commands are issued very rarely, at most one in 32,768 ACTs as analyzed in Section 3.2. Hence,

[①] The newly proposed ARR command can also be directly used by MC to avoid the need to know the row remapping information within the DRAM devices.

we propose to have an RCD return a negative acknowledgment (nack) signal to the master MC when a conflict occurs. We can leverage already existing feedback path indicating that a command from a MC might fail (e.g., alert_n in DDR4 [24]). The RCD can return this signal back to the MC until it finishes the ARR if it receives normal commands to the bank performing ARR. The RCD also sends the nack signal back to the MC while performing an ARR command if there is an ACT command to the rank which includes the bank performing ARR. Because of the additional ACTs performed from ARR, the number of ACTs recognized by the MC and the actual number of ACTs performed in a DRAM rank may differ, which can lead to a violation of the $tFAW$ timing constraint of the DRAM if not careful. Blocking every ACT to the rank during ARR addresses this problem. While the approach is conservative, it has a minimal impact on system performance because the ARR commands are only issued infrequently, at most once when the number of ACTs reaches the RH threshold. The evaluation results in Section 3.6 show that this blocking has no performance overhead except for actual RH attacks because general workloads invoke no ARR. Similar to the case of handling an address signal parity bit error in DDR4, a MC can resend the command that was just blocked.

RH prevention through TWiCe within RCD and ARR interface eliminates the side-channel attacks that use the ACT count information or aggressor and victim row information in TWiCe table.

The processor components, including a MC, cannot access the TWiCe table in an RCD. Also, when there is ARR operation caused by row-hammering, the MC cannot know the aggressor and victim row information because TWiCe sends simply ARR command instead of precharge command, and DRAM calculates the adjacent row addresses.

3.5 Analysis

We analyzed the area, energy, and performance overhead of our proposals using SPICE simulations based on 45 nm FreePDK library [39]. We designed TWiCe as four banks of content addressable memory (CAM) and SRAM. We set *tREFW*, *tREFI*, *tRC*, and th_{RH} as 64 ms, 7.8125 μs, 45.32 ns, and 32,768, respectively. We set th_{PI} and max_act to 4 and 164. Also, we set the number of rows per bank to 131,072.

Area overhead: TWiCe incurs negligible area overhead. Each entry in a TWiCe table needs 46 bits, including (1, 17, 15, 13) bits for ($valid\_bit$, row_addr, act_cnt, $life$). We designed $valid\_bit$ and row_addr as CAM for concurrent searching, and act_cnt and $life$ as SRAM to save area and energy. According to Section 3.3, 553 entries are needed per table, which translates to 3.11 KB per 1 GB DRAM bank.

		Timing (ns)	Energy (nJ)
TWiCe	ACT count	3	0.082
	Table update	140	0.663
DRAM	ACT+PRE (*tRC*)	45.32	11.49
	Refresh/bank (*tRFC*)	350	132.24

Table 3.2. Timing and energy in operating TWiCe and DRAM devices.

Performance overhead: TWiCe incurs no performance overhead while performing TWiCe table updates. TWiCe operations are performed in parallel with normal DRAM activation and auto-refresh operations. Our simulation results show that the count time of TWiCe is 3 ns, which is much less than *tRC* (Table 3.2). We structured TWiCe entries into four banks to reduce the time for table updates. The table update of TWiCe with concurrent access to all banks takes 140 ns and can be performed during an auto-refresh, which takes 350 ns (*tRFC*). For DRAM devices with smaller *tRFC*, we can speed up the table update of TWiCe by populating more banks. In theory, TWiCe may have false positives and issue more ACTs than necessary because th_{RH} is set conservatively. However, the impact of the false positives is negligible in practice because every false positive requires th_{RH} ACTs but incurs mere two additional ACTs as shown in Section 3.6.

Energy overhead: TWiCe requires minimal additional energy as quantified in Table 3.2. As an ACT count operation accompanies

DRAM activation and precharge operations, its overhead of TWiCe is only 0.7% on modern DDR4 [40]. Compared to per-bank auto-refresh energy during *tRFC*, table update overhead is 0.5%. Our analysis is based on 45 nm process; if designed with the latest processes, the energy overhead would be even smaller.

3.6 Evaluation

We evaluated how many additional refreshes TWiCe generates to prevent RH through simulation. We modeled a chip-multiprocessor system by modifying McSimA+ [41] with default parameters summarized in Table 3.3. The system consists of 16 out-of-order cores with a 3.6 GHz operating frequency and 2 memory channels. Each MC is connected to 2 ranks of DDR4-2400 modules and has 64 request queue entries. Each rank has 16 banks. We used DRAM timing parameters and TWiCe thresholds in Table 3.1. We used minimalist-open DRAM page policy [42].

Simulations were run using multi-programmed and multi-threaded workloads. We used the SPEC CPU2006 benchmark suite [43] for multi-programmed workloads. Using Simpoint [44], we extracted and used the most representative 100M instructions per application. We used 29 of SPECrate and 2 of mixed multi-programmed workloads. Each SPECrate workload consists of 16 copies of one application. In order to make the mixed workloads, we measured the memory access per kilo-instructions (MAPKI) of each

application and classified nine most memory intensive applications as spec-high (mcf, milc, leslie3d, soplex, GemsFDTD, libquantum, lbm, sphinx3, and omnetpp). We then made a mix-high multi-programmed workload consisting of the spec-high applications and a mix-blend workload which consists of 16 random SPEC CPU2006 applications regardless of MAPKI. MICA [45] (multi-threaded key-value store), PageRank from GAP benchmark suite [46], and RADIX and FFT from SPLASH-2X [47] were used for multi-threaded workloads.

Resource		Value
Number of cores, MCs		16, 2
Per core	Freq, issue/commit width	3.6 GHz, 4/4 slots
	Issue policy	Out-of-Order
	L1 I/D $, L2 $	16 KB, 128 KB private
	L1, L2 $ line size	64 B
	Hardware (linear) prefetch	On
L3 $ / line size		16 MB shared / 64 B
Per MC	# of channels, Req Q	2 Ch, 64 entries
	Baseline module type	DDR4-2400
	Capacity/rank, bandwidth	16 GB, 19.2 GB/s
	Scheduling policy	PAR-BS [48]
	DRAM page policy	Minimalist-open [42]

Table 3.3. Default parameters of the simulated system.

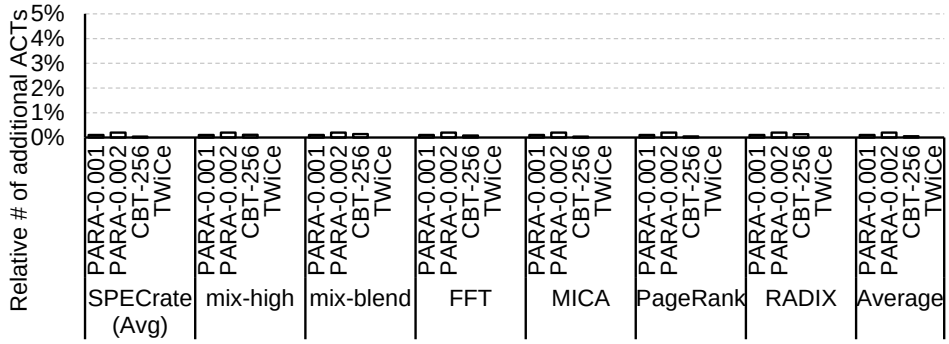
We also used synthetic workloads (S1, S2, and S3) to produce more controlled situations. S1 injects random access sequences

constantly. S2 represents an adversarial memory access pattern for CBT, which keeps accessing a half of entire DRAM rows of a bank until all CBT counters split and then repeatedly accesses the other half after all counters are allocated (described in Section 2.6). S3 is a typical RH attack, which repeatedly accesses only one DRAM row.

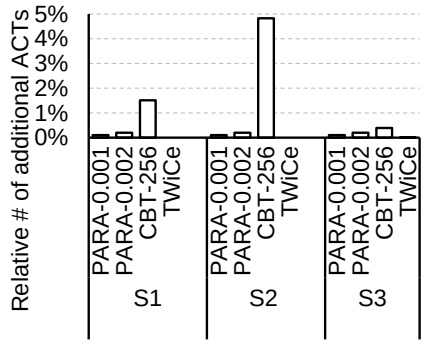
Figure 3.4 shows the relative number of additional ACTs (caused by ARRs in the case of TWiCe) compared to the number of normal ACTs. We compared **TWiCe** with previous solutions. **PARA-0.001** and **PARA-0.002** are PARA refreshing adjacent rows with a probability of 0.001 and 0.002, respectively. **CBT-256** is CBT with 256 counters per bank. We used a threshold of 32K and 11 sub-thresholds for **CBT-256**, the values that were used in evaluating CBT [9].

All solutions generate less than 0.3% of additional ACTs to prevent RH on the evaluated multi-programmed and multi-threaded workloads. Because the memory access patterns of these workloads do not actually cause an RH attack, the additional ACTs on these workloads are due to false positives. **TWiCe** generated no additional ACTs on all multi-programmed and multi-threaded workloads. **PARA-0.001**, **PARA-0.002**, and **CBT-256** produced additional ACTs of 0.1%, 0.2%, and 0.05% on average, respectively.

TWiCe also rarely generates additional ACTs on the synthetic workloads. It only generates additional ACTs of 0.006% on S3, and



(a) Multi-programmed and multi-threaded workloads



(b) Synthetic workloads

Figure 3.4. The relative number of additional ACTs of PARA-0.001, PARA-0.002, CBT-256, and TWiCe compared to the number of normal ACTs on multi-programmed and multi-threaded workloads (multi-programmed SPEC CPU2006, multi-threaded SPLASH-2X, GAP-BS, and MICA applications) and synthetic workloads (S1, S2, and S3). TWiCe does not incur additional ACTs on the multi-programmed, multi-threaded, S1 and S2 workloads and incurs only 0.006% additional ACTs on S3 (RH attack scenario) workload. PARA-0.001 and PARA-0.002 produce additional ACTs of 0.1% and 0.2% on average, respectively. CBT-256 generates up to 4.82% additional ACTs on S2 workload.

still does not make additional ACTs on S1 and S2. PARA-0.001 and PARA-0.002 shows 0.1% and 0.2% additional ACTs on S1, S2 and S3, respectively. By contrast, CBT-256 generates additional ACTs much more frequently on these synthetic workloads. Especially on S2 whose access pattern is adversarial to CBT in particular, it requires additional ACTs of 4.82%. For S3, which represents an RH attack pattern, CBT-256 requires 0.39% of additional ACTs. Because the number of rows that the last level (level 11) counter in CBT-256 should track is $131{,}072/2^{11-1} = 2^{17}/2^{10} = 128$, it has to refresh 128 rows for every 32K ACTs. Therefore, the frequency of false positive detection by TWiCe is orders of magnitude lower than that by the previous RH prevention schemes on adversarial memory access patterns.

Chapter 4.

Optimizing TWiCe to Reduce Implementation Cost

Original TWiCe, which proposed in Chapter 3, requires 553 table entries per bank at 128k of N_{th}. Considering that the overall main-memory system is composed of multiple banks, the required number of table entries is proportion to the number of DRAM banks, increasing the implementation cost of TWiCe accordingly. In this chapter, we propose and evaluate various methods to reduce the implementation cost of TWiCe.

4.1 Pseudo-associative TWiCe

A straightforward implementation of proposed TWiCe would be making the table fully associative (fa-TWiCe) using content-

This Section is based on [1, 2]. – © 2019 ACM, and IEEE 2018.
Reprinted, with permissions from ISCA '19, and CAL '18.

addressable memory (CAM). The fully-associative implementation is feasible as the minimal interval between counter updates is dozens of nanoseconds, and the update is not in the critical path of DRAM accesses. Still, in the case of TWiCe against RH, a more energy-efficient implementation is desired compared to fa-TWiCe with 553 ways. A set-associative design looks appealing at first glance, but it suffers from performance degradation for access patterns that thrash sets because a row that is being evicted from the table needs to trigger refreshes for security.

We address this problem by leveraging a pseudo-associative cache design [13] and call it pseudo-associative TWiCe (pa-TWiCe). Each DRAM row is mapped to a preferred set of pa-TWiCe (see Figure 4.1). A set has set-borrowing (SB) indicators, each counting entries used by another set. For a table with N sets, each set has N-1 SB indicators. pa-TWiCe records a row ACT as follows: 1) it probes the target address in the preferred set. 2) If 1) fails, it checks the non-preferred sets with their SB indicators for the preferred set being non-zero. 3) If the target row is found, the *act_cnt* of that entry is increased by one. 4) If 2) fails, an entry is inserted into a set (preferably to the preferred set) and the corresponding SB indicator is increased by one if needed. When an entry is invalidated, the SB indicator value is decreased by one. pa-TWiCe is inferior to fa-TWiCe in the worst-case for latency and energy efficiency when all the sets must be checked. However,

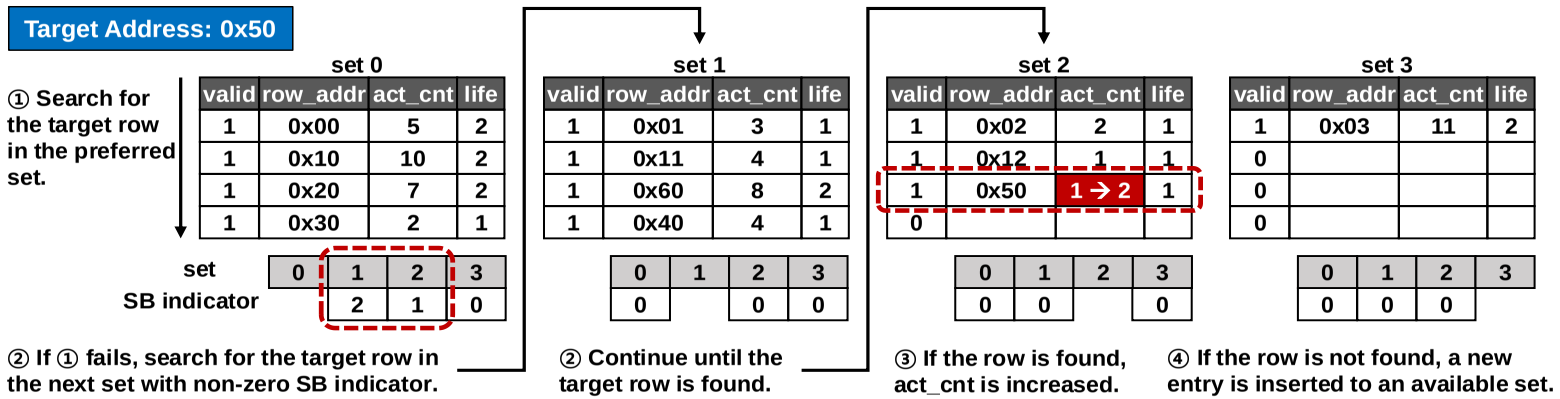


Figure 4.1. Exemplar pseudo-associative TWiCe (pa-TWiCe) operations with a target row address of '0x50' whose preferred set is 0. Each set has set-borrowing (SB) indicators which count the number of entries used by other sets.

because both preferred and non-preferred sets can be checked within tRC, there is no performance overhead. Also, pa-TWiCe can greatly save energy in common cases when checking the preferred set is enough. We show that almost table accesses of the pa-TWiCe target to the preferred set on general workloads in Section 4.5.

4.2 Rank-level TWiCe

The key property of the DRAM interface composing TWiCe is that the number of ACTs to a bank for a specific time (e.g., *tREFW* or *tREFI*) is limited by *tRC*. From this property, we can calculate the required number of TWiCe table entries, which is 553 per bank under the DRAM timing parameters of Table 3.1. In this case, a typical memory system with multiple DRAM banks requires total (553 × the number of banks) table entries.

Managing TWiCe table at a DRAM rank level reduces the number of table entries required. Each bank within a rank can operate independently, but there are *tRRD* (Row to Row Delay) and *tFAW* (Four Activate Window) timing parameters that limit ACT frequency in rank because ACT operation consumes large currents and over-stress the power delivery network of the device. *tRRD* limits the minimum timing of two consecutive ACTs within DRAM devices, and *tFAW* means a time window where four ACTs can be issued. In other words, more than four ACTs within *tFAW* cannot be issued, and it is generally more than 4 × *tRRD*. Therefore, the maximum number of

ACTs within a rank during *tREFI* ($max_{act-rank}$) is $\frac{tREFI-tRFC}{tFAW/4}$, which is less than ($the\ number\ of\ banks\ per\ rank \times max_{act}$); max_{act} is the number of ACTs within a bank during *tREFI* calculated with *tRC*. For example, with the typical DDR4 timing parameter described in Table 4.1, $max_{act-rank}$ is 1,356, while max_{act} is 164. Because the number of banks per rank (device) of DDR4 is 16, $max_{act-rank}$ is almost half of $16 \times max_{act}$ (= 2,624).

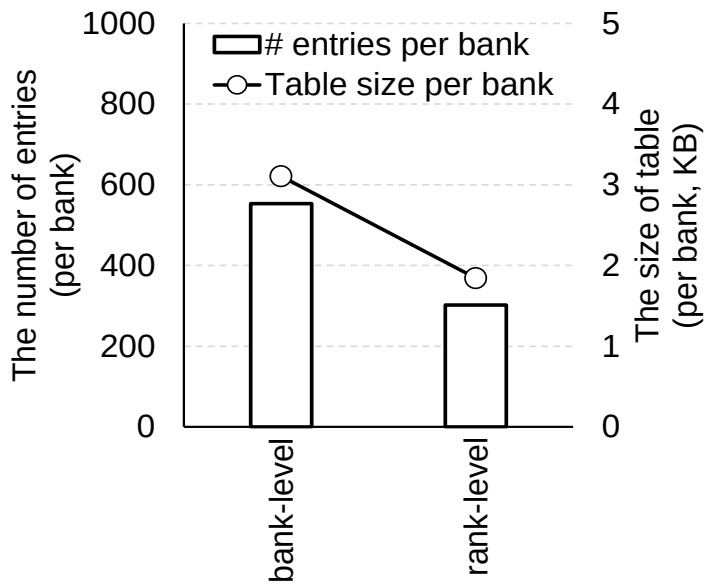


Figure 4.2. The table size comparison between bank- and rank-level TWiCe. The number of entries and the size of table per bank of rank-level TWiCe is 55% and 59% of those of bank-level TWiCe, respectively.

Based on this insight, we propose a rank-level TWiCe, which manages tables in rank-level rather than bank-level. Rank-level

TWiCe can reduce the total number of required table entries. Figure 4.2 compares the average number of entries per bank and the actual table size between bank- and rank-level TWiCe. We calculated the number of required entries based on Section 3.3, and for the rank-level TWiCe, we divided the number of entries of rank with the number of banks per rank. While bank-level TWiCe requires 553 entries per bank, rank-level TWiCe needs 302 entries per bank, which is 55% of bank-level TWiCe. Also, the size of TWiCe table is 3.11 KB and 1.84 KB for bank- and rank-level TWiCe, respectively. The difference of table size between bank- and rank-level TWiCe is a little smaller than that of the number of entries because 4 bits for bank address should be added to *row_addr* of each entry in rank-level TWiCe.

Rank-level TWiCe reduces the number of required table entries per bank, but the total size of the table to be explored is larger than bank-level TWiCe because it is difficult to assign 302 entries to each bank independently. If there are excessive ACTs to a specific bank, it requires still 553 entries for that bank, so all entries in a rank must be managed in an integrated manner. Searching for all of the table entries in a rank increases energy overhead and latency for searching the target row. To reduce these overheads, we implement rank-level TWiCe by exploiting the structure of pa-TWiCe proposed in Section 4.1.

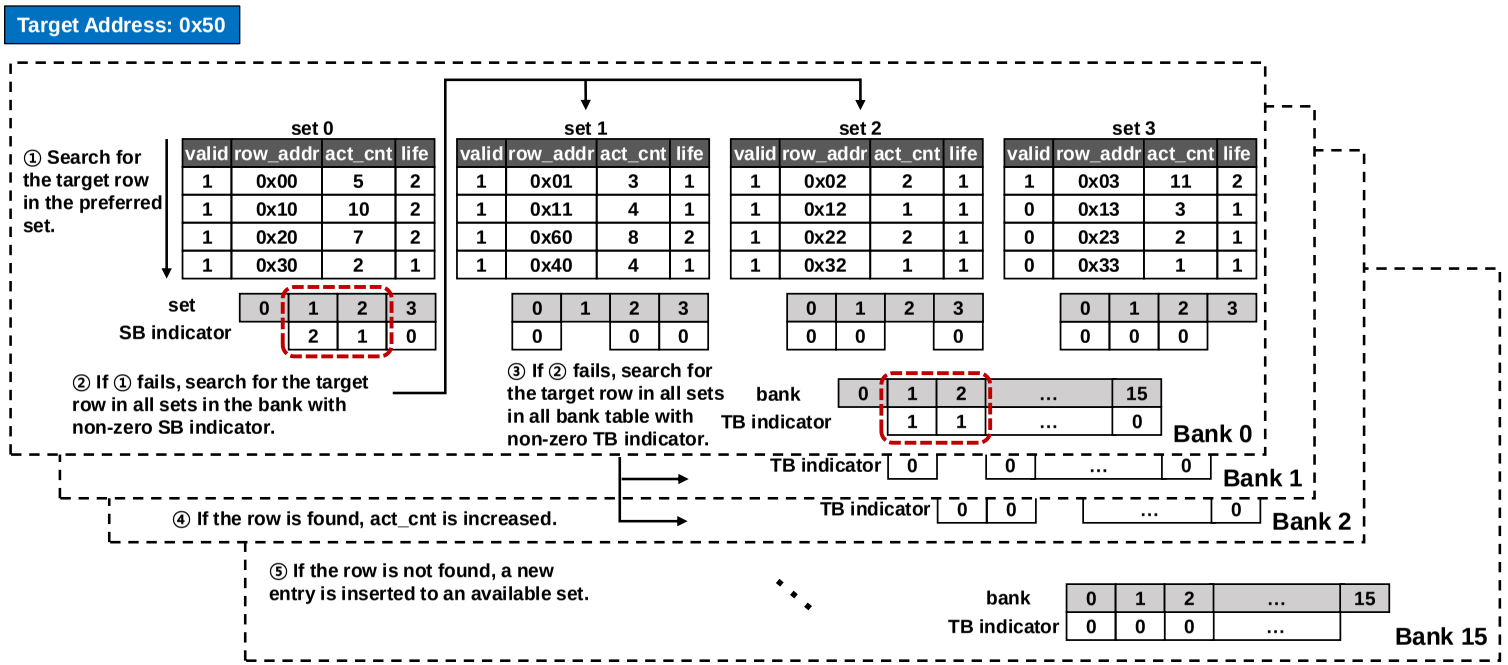


Figure 4.3. The organization of rank-level TWiCe table and exemplar operations with a target row address of '0x50' whose target bank number is 0 and the preferred set is 0. Each set has set-borrowing (SB) indicators that count the number of entries used by other sets, and each bank has table-borrowing (TB) indicators that count the number of entries used by other bank tables.

The structure of rank-level TWiCe is almost identical to bank-level TWiCe. It has an independent table for each bank and a preferred set for a particular target row address. It also has SB indicators, each counting entries used by another set in the bank table. However, if all the table entries for a particular bank are in use, rank-level TWiCe allows borrowing another bank table within the rank. For this, a bank table has table-borrowing (TB) indicators, each counting entries used by another bank table. Rank-level TWiCe counts a row ACT as follows (see Figure 4.3): 1) it probes the target address in the preferred set. 2) If 1) fails, it checks the non-preferred sets with their SB indicators for the preferred set being non-zero. 3) If 2) fails, it checks the non-preferred tables with their TB indicators for the preferred bank being non-zero. 4) If the target row is found, the *act_cnt* of that entry is increased by one. 5) If 3) fails, an entry is inserted into a set (preferably to the preferred table and the preferred set), and the corresponding TB and SB indicators are increased by one if needed. When an entry is invalidated, the TB and SB indicator values are decreased by one.

In a rank-level TWiCe, we need to consider more carefully the worst-case latency where all the bank tables must be checked. Unlike bank-level TWiCe that needs to complete ACT count operation in *tRC*, rank-level TWiCe have to complete ACT count operation in a quarter of *tFAW*; ACT-to-ACT interval to a particular bank is still *tRC*. To satisfy timing constraints, in 2) and 3), it

searches the sets and tables in parallel. In this case, if the target row is stored in a non-preferred set or another bank table, table search consumes more considerable energy than that of bank-level TWiCe. However, there is no significant impact on the actual system because almost all target rows are found in or inserted into preferred sets on general workloads according to our evaluation in Section 4.5.

4.3 Adjusting Threshold to Reduce Table Size

We can reduce the TWiCe table size further by adjusting the threshold of TWiCe. As described in Section 3.2, RH prevention of TWiCe is demonstrated by the following inequality.

$$count_{not-tracked} < th_{PI} \times \frac{tREFW}{tREFI} \quad (1)$$

$$count_{tracked} < th_{RH} \quad (2)$$

$$count_{combined} < th_{PI} \times \frac{tREFW}{tREFI} + th_{RH} < \frac{\mathrm{N_{th}}}{2} \quad (3)$$

The original TWiCe sets th_{PI} to $\frac{th_{RH}}{tREFW/tREFI}$ and makes the right side of the inequality (1) as th_{RH}. Thus, it guarantees RH prevention if th_{RH} is smaller than $\frac{\mathrm{N_{th}}}{4}$. However, the relationship that th_{PI} is $\frac{th_{RH}}{tREFW/tREFI}$ is not essential. Therefore, we can adjust th_{PI} and th_{RH} under conditions that satisfy the inequality (3). In this case, if th_{PI} increase, th_{RH} has to decrease, while th_{RH} can increase when th_{PI}

decreases.

th_{PI}	th_{RH}	Required # of entries (per bank)	
		Bank-level TWiCe	Rank-level TWiCe
1	57,344	1,732	940
2	49,152	946	514
3	40,960	683	373
4 (default)	32,768	553	302
5	24,576	457	250
6	16,384	392	215
7	8,192	339	186

Table 4.1. The required number of TWiCe table entries according to th_{PI} and th_{RH}. We assume N_{th} as 128k. While the original bank-level TWiCe whose th_{PI} is 4 requires 553 entries per bank, rank-level TWiCe with th_{PI} of 7 requires 186 entries.

The threshold adjustment changes the number of required table entries. Increasing th_{PI} makes pruning more strictly; each entry must have a higher act_cnt value to survive than before. On the other hand, decreasing th_{PI} means the entry can survive with a lower act_cnt value than before. Table 4.1 shows the number of required table entries under various th_{PI} value according to the calculation of Section 3.3. Increasing th_{PI} to 7 can reduce the number of required entries of rank-level TWiCe to 186 per bank. This is almost 1/3 of the number of required entries of bank-level TWiCe with th_{PI} of 4 what we first proposed. However, increasing th_{PI} induce more false-positive detections due to smaller th_{RH}. Therefore, in Section

4.5, we experiment with the increase of false-positive detection due to threshold adjustment, and our analysis shows that it is insignificant in general workloads.

4.4 Analysis

We analyzed the area, energy, and timing overhead of threshold adjusted rank-level TWiCe using SPICE simulations based on 45 nm FreePDK library [39]. We designed our proposal as 64-way SRAM. We set *tREFW*, *tREFI*, and *tFAW* to 64 ms, 7.8125 us, and 21 ns, respectively. Also, we set th_{RH}, th_{PI}, and the number of rows per bank to 8,192, 7, and 131,072, respectively.

Threshold adjusted rank-level TWiCe incurs less area overhead than the original TWiCe. Each entry needs 46 bits, including (1, 21, 13, 11) bits for ($valid\_bit$, row_addr, act_cnt, $life$). When compared to original TWiCe, row_addr increases 4 bits due to bank address, act_cnt decreases 2 bits due to the reduction of th_{RH}. Also, $life$ is 11 bits, which is 2 bits smaller than that of original TWiCe, because the maximum $life$ of the entry with act_cnt of 8,191 is $\frac{8,191}{7} = 1,170$. According to Table 4.1, rank-level TWiCe, whose th_{PI} is 7, requires 186 entries per bank, so each bank table is comprised of three sets of 64-way SRAM, which translates to 1.08 KB per 1 GB DRAM bank.

Rank-level TWiCe incurs no performance overhead while performing table updates. Our simulation results show that Rank-

level TWiCe requires 3 ns for accessing a single counter set and 5 ns for all bank tables, which is shorter than $\frac{tFAW}{4}$. Also, it requires 130 ns for updating the table, while the auto-refresh operation of DRAM takes 350 ns. In addition, rank-level TWiCe achieves lower energy overhead than the original TWiCe because it reduces the required number of entries and consists of SRAM instead of CAM. Although the energy required to search all bank tables is 0.861 nJ, which is more significant than that of the ACT count of original TWiCe, we found that the counters for all rows remained in their preferred sets through the multi-programmed and multi-threaded workload simulations specified in Section 4.5.

		Timing (ns)	Energy (nJ)
Original TWiCe	ACT count	3	0.082
	Table update (per bank)	140	0.663
Rank-level TWiCe	ACT count (preferred set)	3	0.018
	ACT count (preferred bank table)	4	0.054
	ACT count (all bank tables)	5	0.861
	Table update (per bank)	130	0.153
DRAM	ACT to ACT (tFAW/4)	5.5	11.49
	Refresh/bank (tRFC)	350	132.24

Table 4.2. Timing and energy in operating original and rank-level TWiCe, and DRAM devices.

4.5 Evaluation

We first evaluated how many target row addresses of ACT are stored in a non-preferred set or another bank table instead of the preferred set. We modeled a chip-multiprocessor system by modifying McSimA+ [41] with the same parameters as Section 3.6. We used DRAM timing parameters in Table 4.1. We used th_{PI} and th_{RH} of 7 and 8,192, respectively.

We run simulations using multi-programmed and multi-threaded workloads. The workloads used for the simulation is the same as in Section 3.6. We used mix-high, mix-blend, and 29 of SPECrate multi-programmed workloads. Also, we used MICA [45], PageRank from GAP benchmark suite [46], and RADIX and FFT from SPLASH-2X [47] for multi-threaded workloads.

Figure 4.4 shows how many target row addresses of ACT are stored in a non-preferred set or another bank table instead of the preferred set. We compared the results of the various number of ways in rank-level TWiCe. The table accesses in the graph includes both access for searching and access for inserting a new entry.

When using SRAM with 32 or more ways, the rank-level TWiCe always finds the target address in the preferred set on the evaluated multi-programmed and multi-threaded workloads. With 16-way SRAM, three SPECrate workloads make target row address searching (or inserting) in the non-preferred set, but it was smaller

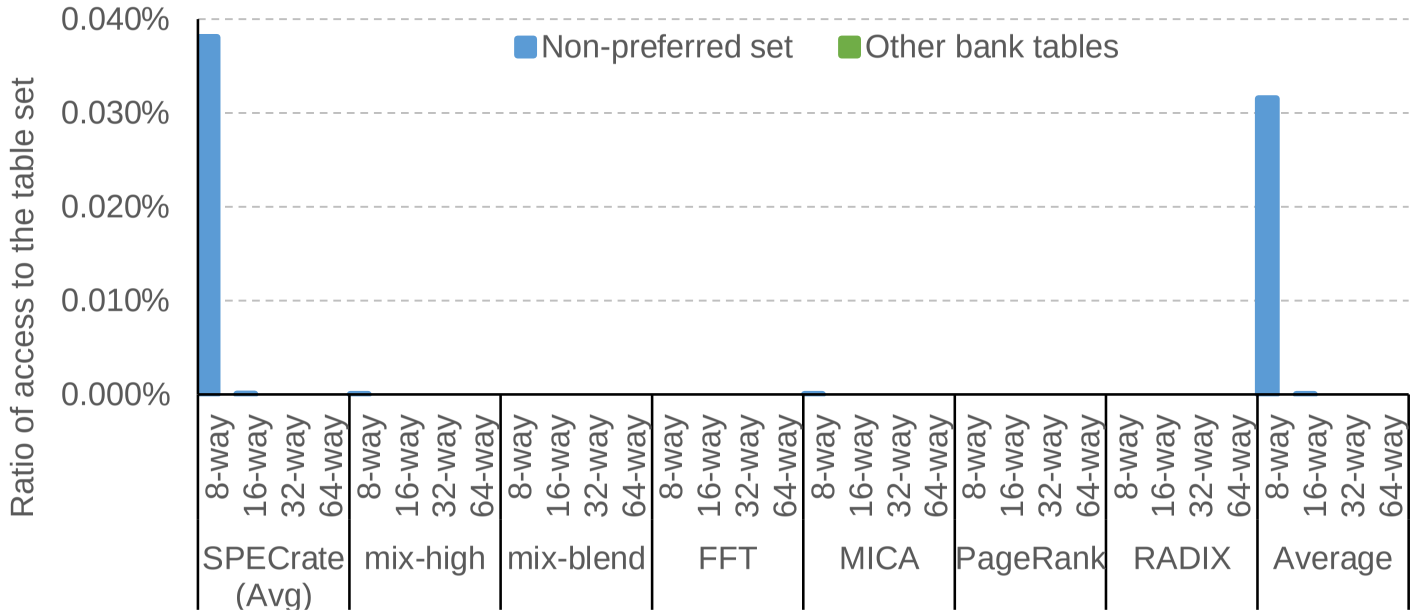


Figure 4.4. The ratio of access to non-preferred set and other bank tables on multi-programmed and multi-threaded workloads (multi-programmed SPEC CPU2006, multi-threaded SPLASH-2X, GAP-BS, and MICA applications). When using 32 or more ways, every TWiCe table access only heads to the preferred set.

than 0.001% of the total number of ACTs. When using 8-way SRAM, 0.03% of the total table accesses head to non-preferred sets. However, even in this case, there is no other bank table access.

We also evaluated how many additional refreshes (ACTs) rank-level TWiCe generates to prevent RH through simulation. In addition to the workloads used above, we used three synthetic workloads (S1, S2, and S3) that are the same as Section 3.6. Rank-level TWiCe generates no additional ACTs on the evaluated workloads except for S3 synthetic workloads. Even on S3 workloads, it only generates additional ACTs of 0.024%, which is four times of that of original TWiCe. It is still much smaller than 0.1% of PARA-0.001.

Chapter 5

Augmenting TWiCe for Hot-page Detection

TWiCe, which is proposed to prevent RH, contains information about the frequently activated rows. In this section, we augment TWiCe for hot-page detection in the memory system consisting of asymmetric latency DRAM.

5.1 Necessity of Counters for Detecting Hot Pages

In a conventional DRAM device, each timing parameter is set to the worst-case latency, such as the latency to access the farthest cell in topological distance from the I/O pins. By partitioning a device into multiple regions, there is a room to reduce access latency on a subset of these regions. Numerous studies [7], [14], [17], [49], [50], [51], [52] have proposed main-memory DRAM microarchitectures that support asymmetric access latency. In

particular, a few of them [7], [14], [17] have focused on the fact that key memory access latency values, such as *tRCD* and *tRP*, can be reduced if the length of the DRAM BLs is shortened. These studies divide DRAM internal structure into a fast access region with shorter BLs and a slow access region with longer BLs.

Tiered-Latency DRAM (TLDRAM [14]) divides each BL within subarrays into two short BLs through an isolation (ISO) transistor. TLDRAM has two types of rows: one is near row which is always connected to the corresponding BLSA, and the other is far row which is located farther from BLSA than the near rows and is connected through an ISO transistor. When a near row is accessed, latency can be shortened by disconnecting the ISO transistor, which reduces the BL capacitance. The average access latency can be reduced by allocating frequently accessed data in the near rows.

Center high aspect ratio mat (CHARM [7]) architecture places high-aspect-ratio (HAR) mats that have shorter timing parameters in the center (closer to I/O pads) area, and normal mats that have default timing parameters in the remaining area. As the BL capacitance of a HAR mat is smaller than that of a normal mat, the latency and power consumption on an access to the HAR mat is reduced. Also, column access latency (*tCL*) of the center area is further reduced by shifting the per-bank column decoder in the center area closer to the I/O pads.

Main-memory accesses in many workloads are concentrated to

a small portion (hot pages) of their entire memory footprints. Therefore, exploiting asymmetric low-latency main memory and allocating frequently accessed hot pages to the fast region would help improving system performance. However, it is difficult to identify hot pages because hot pages can change dynamically at runtime. Misidentifying hot pages may even harm performance. In finding hot pages, static allocation is suboptimal because it requires off-line profiling to extract memory access patterns, which can vary significantly depending on the input values and the execution phases of running applications as well as interaction with other processes in a system. Dynamic allocation can solve these problems but it requires real-time hot-page detection mechanism with high accuracy and a low-latency page swap mechanism. Although the operating system (OS) can track page access distribution, this information is not appropriate to use for asymmetric latency DRAM because it does not consider the cache hierarchy. The hot page determined by OS requires little DRAM accesses because the accesses to this page mostly hit in the cache. Therefore, there have been several studies on how to identify hot pages [7], [17], [53], [54].

5.2 Previous Studies on Migration for Asymmetric Low-latency DRAM

There have been multiple DRAM microarchitecture proposals targeting high-throughput internal DRAM data transfer, which can be

used to relocate frequently accessed data to the fast region of asymmetric DRAM [15], [16], [17]. Chang et al. [16] proposed inter-linked subarrays (LISA) which enable fast data movement between subarrays. LISA inserts ISO transistors to connect BLs and a row-buffer that are adjacent but not connected. By turning the ISO transistors on, data can be copied between subarrays. DAS-DRAM [17] also enables rapid data movement between subarrays. To reduce the overhead of row migration, DAS-DRAM adds a 2T2C migration cell to each BL and use the migration cells as temporal buffers needed for the swap. DAS-DRAM achieves 3×*tRC* latency for swapping two rows (*tSWP*) that are located in adjacent subarray by concurrently utilizing the migration cells in the both subarrays. We leverage DAS-DRAM microarchitecture for DRAM page swap as it does not require a spare row for swapping. Compared to DAS-DRAM which swaps a DRAM page after a few accesses, we utilize the counters in TWiCe to choose page-swap targets.

Figure 5.1 shows our asymmetric DRAM architecture adopting the migration cells in DAS-DRAM. We pair a low-latency (fast) subarray with short BLs, each connecting 1/3 of DRAM cells compared to a BL in a normal (slow) subarray. One row in the fast region and three rows in the slow region form a row group; a hot page in the slow region can be swapped with the page in the fast region, which was possibly hot in the past. We summarized the timing parameters of this architecture in Table 5.1. The fast region has

reduced values for *tRCD* and *tRP*, but its *tRC* stays unchanged as it influences the number of counters TWiCe needs for row-hammering protection. Note that system performance is more sensitive to *tRCD* and *tRP* than *tRAS* in modern servers with hundreds of DRAM banks because a bank typically stays deactivated (because they employ an adaptive-open page management policy [55]) or services a sequence of column accesses to an activated row (for access patterns with spatial locality) and suffers less frequently from row-buffer conflicts.

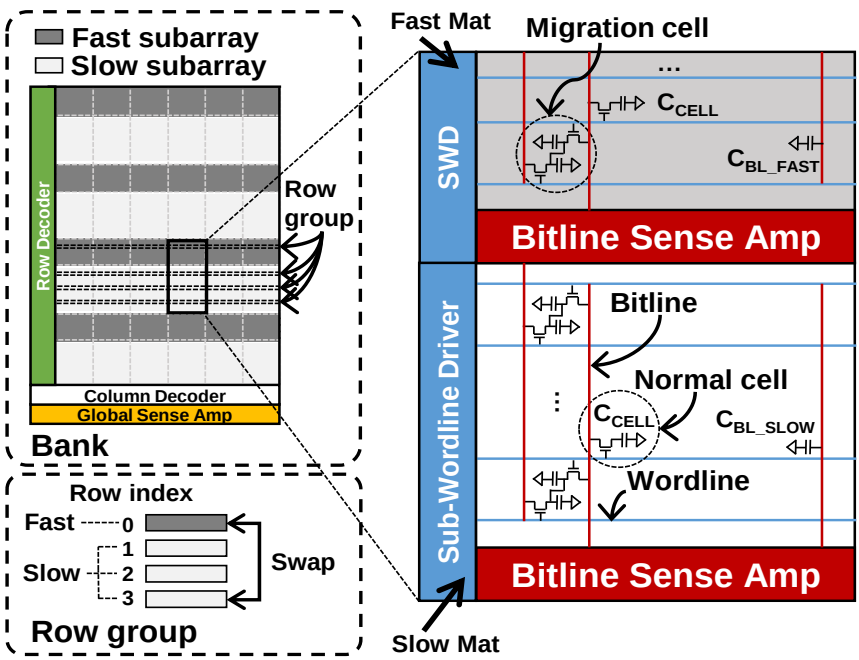


Figure 5.1. Low latency DRAM microarchitecture based on dynamic asymmetric subarray DRAM (DAS-DRAM [17]). A row in a slow subarray can be swapped with a row in a fast subarray through migration cells.

5.3 Extending TWiCe for Dynamic Hot-page Detection

Because TWiCe tracks rows that are activated recently or frequently, we leverage this information to detect hot pages to be placed in the fast DRAM region. We argue that hot-page detection with ACT counts is reasonable based on the following rationale: (1) Recently accessed rows are more likely to be accessed again because of temporal locality in memory accesses [17], [49]. (2) In a system running many applications concurrently, a large portion of main-memory accesses accompany ACT due to inter-core interference in shared memory [56] or adaptive-open page management policy, which diligently deactivates rows that are idle for more than a certain (short) period [42], [55]. (3) The low-latency DRAM microarchitectures, such as DAS-DRAM [17], focused on reducing *tRCD* and *tRP*, which affects the latency of PRE/ACT. Compared to access-based hot-page detection, ACT-based hot-page detection filters out accesses with frequent row-buffer hits, which have the same latency in both fast and slow regions, reducing ineffective page swaps which gain little with the shortened timing parameters of the fast region. Therefore, hot-page detection through TWiCe ensures that hot pages are migrated to the fast region of DRAM with high accuracy and fewer row swaps, as quantitatively evaluated in Section 5.5.2.

We augment TWiCe to detect hot pages (see Figure 5.2). Our asymmetric DRAM architecture has two DRAM addressing types as a DRAM row could be swapped within a row group. One is physical row address specified by a request from the MC, and the other is device row address indicating the location within a DRAM device. To track row swap record, TWiCe has an address translation table, which is used to translate a physical address to a device address.

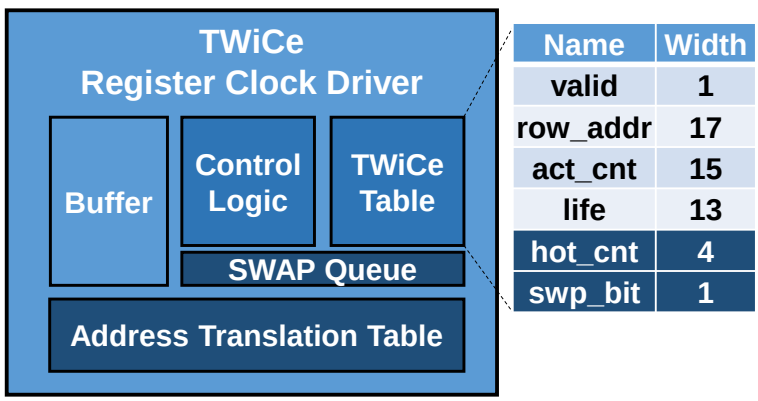


Figure 5.2. Extended TWiCe microarchitecture with additional *hot_cnt*, *swp_bit*, swap queue, and address translation table.

The count information (*act_cnt*) of swapped rows does not represent hot-page information because of the mismatch between physical and device row address. TWiCe manages the entries based on the device row address, indicating the actual location on a device to prevent row-hammering. Therefore, after rows are swapped, *act_cnt* of each entry of the swapped rows cannot represent whether the physical row mapped to that entry is hot. To solve this problem, we add a new saturating counter called *hot_cnt* to each entry of

TWiCe. Also, we define a new hot-page detection threshold (th_{HP}). hot_cnt counts the ACTs to the device row address of each entry, and the row becomes a swap candidate if hot_cnt reaches th_{HP}. As opposed to act_cnt, hot_cnt resets to zero when the row is swapped so that the ACT count for the swapped physical row address is reinitialized. When a row is swapped, hot_cnt of all rows in the row group to which the swapped row belongs is set to zero to prevent excessive swaps due to competition between the multiple hot rows within a row group.

We also add swp_bit to each TWiCe entry to prevent consecutive swaps to a row group. It indicates that the row has been swapped in the preceding pruning interval (PI). swp_bit holds one for a PI after a row swap, and if swp_bit of the target row is one, the row is not swapped even if it is detected as a hot page. This ensures that the row in the fast region can stay for a certain amount of time ($2\times tREFI$ here), preventing excessive swapping. Finally, in order to manage the row swap candidates, we add a swap queue which contains the source and target device row pairs to swap.

The process of detecting hot pages and placing them to the fast region of asymmetric DRAM with TWiCe is as follows: 1) When ACT is sent to the target physical row, its address is translated to the corresponding device row, and hot_cnt is incremented by one like act_cnt. 2) If hot_cnt of the entry reaches th_{HP} and is in the slow region, TWiCe searches the fast region row of the row group (target

row) in TWiCe table. If the swp_bit of the target row entry is zero or the target row entry does not exist, source and target row address is inserted into a swap queue entry unless the target row is already in the swap queue. 3) If hot_cnt of the entry reaches th_{HP} and is in the fast region, its swp_bit is set to one, preventing this hot page from being swapped in the current PI. Also, if this row is the target row of a swap queue entry, that swap queue entry is removed as we cannot decide which rows are hotter. 4) When the TWiCe table is updated during auto-refresh, TWiCe reads the swap queue and finds the rows in TWiCe table which belongs to the same row group, and sets their hot_cnt to zero. Also, act_cnt of the source and target entry is increased by one because row swap incurs additional activations to both. If the target row entry does not exist in TWiCe table, it is newly inserted. Also, swp_bit of the target entry is set to one to disable swap in the next PI. swp_bit of all entries that did not migrate in the current PI is set to zero, enabling swap in the next PI. 5) After the auto-refresh, the source and target row address pairs in the swap queue are swapped in turn, and the finished entries are erased from swap queue.

5.4 Additional Components and Methodology

Address translation is essential in asymmetric memory systems supporting hardware managed migration; however, as opposed to previous proposals which place a (cache of) translation table within

a MC [17], [57], [58], our address translation table is located at RCD. In order to reduce the size of the address translation table, we design it with Lehmer code encoding [59] and store the permutation of the rows in a row group instead of one-to-one mapping of device row address over each physical row address. In our design, as a row group consists of four rows, the number of possible permutation is 4! = 24. Therefore, each row group stores 5-bit encoded data that can represent 24 permutations, resulting in 1.25 bit per row. A DDR4 16 GB DIMM populated with two ranks, which is used for our evaluation (Section 5.5.2), needs 307 KB for an address translation table. The encoding/decoding overhead of address translation is described in Section 5.5.1.

As described in Section 5.3, row swap is performed right after auto-refresh. During auto-refresh, TWiCe in RCD sends source-target row address pairs in the swap queue to the corresponding banks in the DRAM devices (each bank holds the swap row address info; 17 bits for source ***row_addr*** and 2 bits for target row in a row group = 19 bits per swap). After auto-refresh, a row swap is executed at a fixed time, $tSWP$ ($3 \times tRC$ as mentioned in Section 5.2).

To handle swap timing, TWiCe can send two additional information to MC through an alert signal path. First, when row swap is in progress and the RCD receives a new ACT from the MC, TWiCe sends a signal indicating that row swap is not yet finished. If the MC receives this signal, it reschedules to send the next command after

tSWP. If there are several entries in swap queue, the time is increased by *tSWP* per entry. Second, TWiCe sends another signal to adjust timing parameter (*tRCD* and *tRP*) if the currently accessed row is in the fast region. MC normally accepts the timing as the slow region of DRAM, but when MC receives this signal, MC reduces *tRCD* or *tRP* counter, respectively, as much as the timing difference between the slow and the fast region. We analyze the latency of adjusting the timing parameters in Section 5.5.1.

We also add a parameter, max_{swp}, which limits the maximum number of swaps possible per refresh command (which is equal to PI) to mitigate excessively long swap latency. Because swap latency is extended by *tSWP* per swap on a bank, system performance can be degraded due to delayed memory accesses if there are a lot of swaps to be processed, which effectively increases *tRFC* from the access scheduler's perspective within a MC. To alleviate this problem, we implement max_{swp} by adjusting the size of the swap queue. When the swap queue is full, no more swap is enqueued. A small max_{swp} size can reduce performance degradation due to swap latency, but if it is too small, TWiCe cannot follow a rapid change in hot pages as the number of hot pages that can be moved to the fast region is limited per PI. To analyze this trade-off, we conducted a simulation to observe the sensitivity of max_{swp} on performance using the experimental setup in Section 5.5.2. We saw performance improvement becomes much less sensitive when max_{swp} was four or

more. Therefore, we fixed max_{swp} as four in our evaluation.

5.5 Analysis and Evaluation

5.5.1 Overhead Analysis

Our asymmetric DRAM architecture incurs area overhead due to high aspect ratio subarray which has short bitlines and migration cells. As described in Section 5.2, a quarter of the total DRAM capacity is composed of 3× high aspect ratio subarrays and 2 migration cells are needed for each bitline, resulting in about total 7% DRAM area overhead [17], [60].

For hot-page detection, 4-bit *hot_cnt* and 1-bit *swp_bit* are added to each entry of TWiCe, leading to 0.28 KB increase in the table size. TWiCe requires an address translation table and control logic to support swapping, for which an SRAM with 384 KB [61] can be implemented within 2 mm^2 and consumes 180 pJ per access. Maximum operating frequency is 1.2 GHz, which meets target frequency specification in Section 5.5.2. We synthesized TWiCe control, swap queue, Lehmer encoder and decoder for address translation table logics with Synopsys Design Compiler [62] and IC Compiler [63]. Total silicon size is estimated to be less than 0.05 mm^2 at 1.2 GHz.

TWiCe needs a return path to a MC to send alert signals for row-hammering, swapping, and *tRCD* timing adjustment. Therefore, we

need extra pins per MC to enable bidirectional communication between TWiCe and MCs. We resolve this problem by leveraging a currently existing pin in DDR4 called ALERT_n, which is for sending ECC exception. Originally, this pad is designed as an open drain pad for accepting ALERT_n signals from all DRAM chips. However, the load of this signal is greatly decreased by RCD and it can operate at high speeds comparable to the transfer rate of a DQ (data I/O) bus. We modify ALERT_n as a single-ended unidirectional CMOS output [64]. This signal encodes the original ECC error detection as well as three (row-hammering, swapping, *tRCD* timing adjustment) TWiCe functions.

Because the address translation table is placed at RCD, MC does not know if the DRAM row to activate is currently located at the fast region. Therefore, MC should first assume that all DRAM row activation takes *tRCD* of the slow region and later update the *tRCD* value once it receives the information through ALERT_n that the row being activated belongs to the fast region. MC must receive this information within the *tRCD* of the fast region to avoid any performance penalty. We can break down the sequence of this information delivery as follows: ① ACT is sent from MC to RCD. ② The address translation table is accessed. ③ If the row address corresponds to the fast region, the signal is encoded and leaves ALERT_n. ④ The signal is delivered from RCD back to MC. ⑤ The *tRCD* counter value in MC should be decreased accordingly. Here, ①

and ④ are determined by the channel propagation latency between MC and RCD. The transfer latency of MC and DIMM is calculated to be 1 ns per 6 inches, and the required length from MC to DIMM is less than 2 inches [65], [66]; and hence the maximum latency for both ① and ④ should be 0.34 ns. ② and ③ require 1 *tCK* (DRAM cycle time, which is 0.83 ns for DDR4-2400) to access the address translation table and 4 *tCK* for transmitting the encoded TWiCe signal. ⑤ needs 1 *tCK* latency. Because a total latency (<7 *tCK*) is smaller than *tRCD* of fast region (9 *tCK*, see Table III), *tRCD* can be adjusted successfully within MC.

5.5.2 Evaluation

We simulated a chip multi-processor to evaluate the effect of hot-page detection using TWiCe in asymmetric DRAM microarchitecture. We modified McSimA+ [41] with default parameters summarized in Table 5.1. The system consists of 16 out-of-order cores with 3.6 GHz operating frequency and 4 memory channels. Each MC connects to 2 ranks of DDR4-2400 modules and has 64 request queue entries. Each rank has 16 banks and the capacity per rank is 16 GB. We used (16, 16) and (9, 9) *tCK* as (*tRCD*, *tRP*) timing parameters for the slow and fast regions of DRAM, respectively. The page swap latency (*tSWP*) is 165 *tCK* which is $3 \times tRC$. We set default hot-page detection threshold (th_{HP}) to 16 based on the result of sensitivity study in the last of this section and the maximum number of swap per

auto-refresh (max_{swp}) to 4.

Resource		Value
Number of cores, MCs		16, 4
Per core	Freq, issue/commit width	3.6 GHz, 4/4 slots
	Issue policy	Out-of-Order
	L1 I/D $, L2 $	16 KB, 128 KB private
	L1, L2 $ line size	64 B
	Hardware (linear) prefetch	On
L3 $ / line size		16 MB shared / 64 B
Per MC	# of channels, Req Q	4 Ch, 64 entries
	Baseline module type	DDR4-2400
	Capacity/rank, bandwidth	16 GB, 19.2 GB/s
	Scheduling policy	PAR-BS [48]
	DRAM page policy	Minimalist-open [42]
DRAM Timing	*tRCD*, *tRP*	slow region (16, 16) *tCK* fast region (9, 9) *tCK*
	tSWP (swap latency)	165 *tCK*
	th_{HP}, max_{swp}	16, 4

Table 5.1. Default parameters of the simulated system.

Figure 5.3 shows the performance (IPC) improvement of systems employing various hot-page detection schemes including TWiCe on low-latency DRAM microarchitecture depicted in Figure 5.1. For single-threaded workloads, we use a single memory channel to stress main-memory bandwidth. The baseline configuration uses

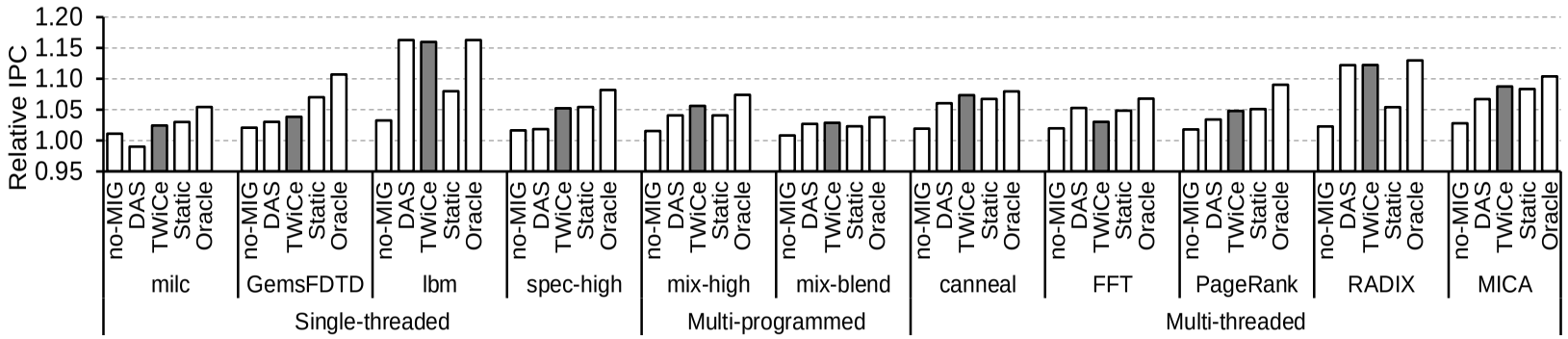
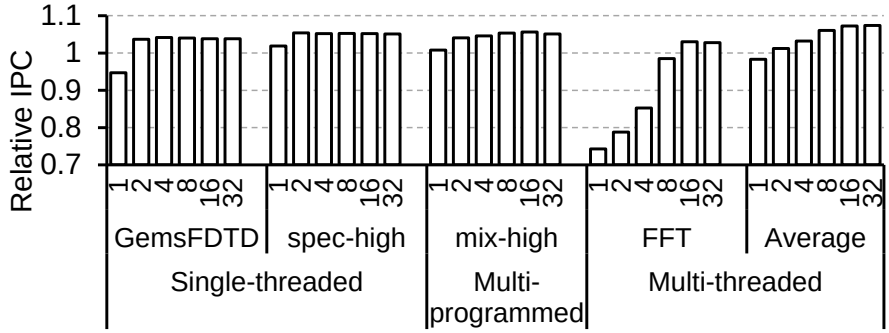


Figure 5.3. The relative IPC of no-MIG (asymmetric memory without swap), DAS (a row-swap scheme in Dynamic Asymmetric Subarray [34]), Static (statically assigning hot pages to the fast region), TWiCe, and Oracle (all rows in the fast region) compared to the baseline DDR4-2400 on single-threaded and multi-programmed SPEC CPU2006, multi-threaded PARSEC, SPLASH-2X, GAP-BS, and MICA applications.

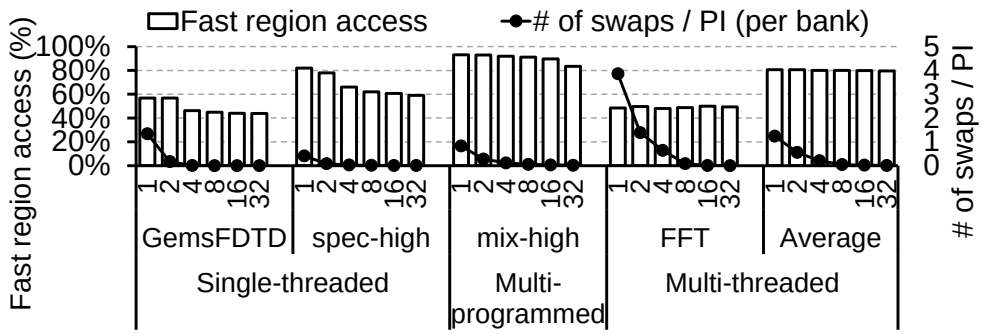
DDR4-2400 DRAM devices without any fast region. In the **no-MIG** configuration, DRAM is divided into fast and slow regions but no migration is performed. In **Static** configuration, the upper 25% of frequently accessed rows are allocated to the fast DRAM region based on offline profiling, without dynamic swapping of DRAM rows. **DAS** is an access-frequency-based DRAM row migration method used in DAS-DRAM [17], which triggers row swapping if a row in the slow DRAM region is accessed eight times. In the **Oracle** configuration, DRAM is hypothetically assumed to have only fast region.

TWiCe outperforms **DAS** and reaches or even surpasses the performance of **Static** in most evaluated workloads. **DAS** selects hot pages by counting the number of accesses to the row, but only the accesses that accompany ACT/PRE operations take the benefit of reduced *tRCD* and *tRP*. Therefore, as **DAS** often prematurely triggers row swapping, its performance is inferior to **TWiCe**. **Static** does not react to variations of hot pages in runtime; this explains why **TWiCe** outperforms **Static** in several workloads. **TWiCe** selects hot pages according to the number of ACT operations within recent pruning intervals (PIs), so it can effectively migrate time-varying hot pages to the fast region. As shown in Figure 5.4(b), **TWiCe** directs 89.7% and 79.9% of total accesses to the fast region at th_{HP} of 16. However, on a few workloads, such as GemsFDTD and FFT, **TWiCe** performs worse than **Static** because the dynamic hot-page tracking mechanism

of TWiCe does not always work ideally; fewer than 50% of total accesses head to the fast region.



(a) The relative IPC



(b) The access ratio to the fast region and the average number of swaps

Figure 5.4. (a) The relative performance (IPC) compared to the baseline DDR4-2400 device, (b) the access ratio to the fast region, and the average number of DRAM row swaps per PI (Pruning Interval) in a bank across a varying number of hot-page detection threshold (th_{HP}).

TWiCe performs on par with Oracle on several workloads even if only 25% of the main-memory capacity TWiCe use belongs to the fast region. TWiCe improves IPC by 5.2% and 5.6% for spec-high

applications and mix-high workloads where the IPC improvements of Oracle are 8.2% and 7.4%. Also, TWiCe achieves 7.4%, 3.0%, 4.8%, 12.2%, and 8.8% IPC improvements over the baseline for canneal, FFT, PageRank, RADIX, and MICA, respectively.

We further conduct a sensitivity study on hot-page detection threshold of TWiCe. Figure 5.4 shows the performance (IPC), the access ratio to the fast region, and the average number of swaps per PI in a bank as we sweep th_{HP} from 1 to 32. The baseline uses DDR4-2400 devices without the fast region.

We made the following key observations. First, the access ratio to the fast region is decreased by increasing th_{HP}, but even when th_{HP} is increased to 32, the ratio is still higher than 50% in most workloads. Figure 5.4(b) shows that on average the access ratio to the fast region is 82% for spec-high, 93% for mix-high, and 80.6% for multi-threaded workloads. The access ratio to the fast region gradually decreases as th_{HP} increases. For th_{HP} of 32, the ratio is 59% for spec-high, 83.5% for mix-high, and 79.6% for multi-threaded workloads. For GemsFDTD and FFT mentioned in the first evaluation, performance improvement is relatively small as only half of total accesses are sent to the fast region.

Second, the average number of swaps per PI is large at low th_{HP}, but sharply decreases as th_{HP} increases. As shown in Figure 5.4(b), at th_{HP} of one, the average number of swaps per PI is 0.41, 0.83, and 1.24 for spec-high, mix-high, and multi-threaded workloads

average, respectively. As number of swaps increases, memory accesses can be delayed due to longer swap latency after auto-refreshes. Therefore, with low th_{HP}, the performance gain due to high access ratio to the fast region is mostly lost as the swap overhead increases steeply. Especially, in the case of FFT, the average number of swaps per PI is almost 4 which results in about 25% performance degradation due to the swap latency as shown in Figure 5.4(a).

As seen from the above observations, as th_{HP} increases, the performance improvement decreases with decreased access ratio to fast region but the swap overhead also decreases with swaps per PI reduction. On multi-programmed and multithreaded workloads, the access ratio loss and swaps per PI reduction are balanced at th_{HP} of 16, with fast region ratio higher than 80% and less than 0.1 swaps per PI on average. Figure 5.4(a) shows that at th_{HP} of 16, TWiCe improves IPC by 5.2%, 5.6%, and 7.2% over baseline for spec-high, mix-high, and multi-threaded workloads, respectively.

Chapter 6

Conclusion

In this thesis, we have proposed TWiCe, a new counter-based hardware solution to combat DRAM row-hammering (RH), and augmented TWiCe for hot-page detection in the low-latency DRAM architecture.

TWiCe precisely tracks the number of ACTs to each DRAM row with a small number of counters and provides strong protection; adjacent rows are guaranteed to be refreshed before the number of ACTs exceeds a RH threshold. The precise protection is possible with low overhead because tracking the number of ACTs only to a small subset of frequently activated DRAM rows is sufficient. To exceed the RH threshold within a refresh window, a row must be frequently activated, but as the total number of DRAM row ACTs

---

This Section is based on [1, 2]. – © 2019 ACM, and IEEE 2018.
Reprinted, with permissions from ISCA '19, and CAL '18.

over a period is limited by the DRAM interface, the maximum number of rows that can be activated frequently, and thereby row-hammered, is bounded. We analytically derive the number of counters that can guarantee precise protection from the RH attack. We distribute the functionality of TWiCe among a MC, RCDs, and DRAM devices, achieving an efficient implementation. Our analysis shows that TWiCe incurs less than 0.7% area/energy overhead on modern DRAM devices and it is free of false positive detection on all the evaluated workloads except no more than 0.006% of additional ACTs on adversarial memory access patterns including RH attack scenarios.

To reduce the area and energy overhead of TWiCe further, we propose threshold adjusted rank-level TWiCe by leveraging a pseudo-associative cache design. Rank-level TWiCe requires a smaller number of table entries because the maximum ACT frequency within a DRAM rank is more bounded than that within a bank. Also, we reduce the number of entries by adjusting TWiCe thresholds. To minimize the impact on performance, we find the appropriate TWiCe thresholds that do not increase the number of false-positive detections on general workloads by simulation.

Finally, we extend TWiCe to improve main-memory performance. TWiCe can be used as a hot-page detector for asymmetric low-latency DRAM microarchitecture, as recently activated pages are likely to be activated again due to temporal locality in memory accesses. We also propose a DRAM row swap

methodology and an address translation table management method with a detailed timing analysis. Counter entries contain hot-page information, and rows whose hot-page activation count exceeds the hot-page detection threshold are swapped with a row in the fast DRAM region. Our evaluation shows that low-latency DRAM using TWiCe achieves up to 5.6% and 12.1% IPC improvement over a baseline DDR4 device for multi-programmed and multithreaded workloads.

6.1 Future work

As described in Chapter 2.4, the row-hammering threshold (N_{th}) is expected to decrease going forward with further technology scaling. To prevent row-hammering with lower Nth, we need the lower TWiCe thresholds (th_{RH} and th_{PI}), increasing the required number of table entries. Even if we apply all optimization techniques in this thesis to TWiCe, more than 1000 entries per bank are required with N_{th} below 10000. Also, according to [68], the activation of the non-adjacent row can cause bit flips. Like double-side RH, TWiCe can resolve this problem by decreasing th_{RH} and th_{PI}, but the table size has to be larger accordingly. Therefore, it would be essential to reduce TWiCe table size further or propose a new low-cost solution.

Bibliography

[1] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. Ahn, "TWiCe: Preventing Row-hammering by Exploiting Time Window Counters," in Proceedings of 46th International Symposium on Computer Architecture (ISCA), June 2019.

[2] E. Lee, S. Lee, G. E. Suh, and J. Ahn, "TWiCe: Time Window Counter Based Row Refresh to Prevent Row-hammering," in IEEE Computer Architecture Letters (CAL), vol. 17, 2018.

[3] I. Bhati, M.-T. Chang, Z. Chishti, S.-L. Lu, and B. Jacob, "DRAM Refresh Mechanisms, Penalties, and Trade-offs," IEEE Transactions on Computers (TC), vol. 65, 2016.

[4] S. Cha et al., "Defect Analysis and Cost-Effective Resilience Architecture for Future DRAM Devices," in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017.

[5] Y. Kim et al., "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in Proceedings of 41st International Symposium on Computer Architecture (ISCA), 2014.

[6] B. Jacob, S. Ng, and D. Wang, "Memory Systems: Cache,

DRAM, Disk," Morgan Kaufmann Publishers Inc., 2007.

[7] Y. H. Son, S. O, Y. Ro, J.W. Lee, and J. Ahn, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," in Proceedings of 40th International Symposium on Computer Architecture (ISCA), 2013.

[8] D.-H. Kim, P. J. Nair, and M. K. Qureshi, "Architectural Support for Mitigating Row Hammering in DRAM Memories," IEEE Computer Architecture Letters (CAL), vol. 14, 2015.

[9] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Mitigating Wordline Crosstalk using Adaptive Trees of Counters," in Proceedings of 45th International Symposium on Computer Architecture (ISCA), June 2018.

[10] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Counter-Based Tree Structure for Row Hammering Mitigation in DRAM," in IEEE Computer Architecture Letters (CAL), vol. 16, 2017.

[11] M. Son, H. Park, J. Ahn, and S. Yoo, "Making DRAM Stronger Against Row Hammering," in Proceedings of the 54th Annual Design Automation Conference (DAC), 2017.

[12] G. Mohsen, L. Mikel, and G. Jim, "A Run-time Memory Hot-row Detector," 2015. [Online]. Available: http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/.

[13] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas, "L1 Data Cache Decomposition for Energy Efficiency," in Proceedings of the 2001 International Symposium on Low Power Electronics and Design (ISLPED), 2001.

[14] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, “Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture,” in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2013.

[15] V. Seshadri et al., “RowClone: Fast and Energy-efficient In-DRAM Bulk Data Copy and Initialization,” in Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2013.

[16] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, “Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-subarray Data Movement in DRAM,” in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016.

[17] S.-L. Lu, Y.-C. Lin, and C.-L. Yang, “Improving DRAM Latency with Dynamic Asymmetric Subarray,” in Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015.

[18] D. Jack et al., "Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake," in IEEE Micro, vol. 37, 2017.

[19] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, "IBM Power9 Processor Architecture," in IEEE Micro, vol. 37, 2017.

[20] T. Singh et al., "Zen: An Energy-Efficient High-Performance × 86 Core," in IEEE Journal of Solid-State Circuits, vol. 53, 2018.

[21] M. Horiguchi and K. Itoh, "Nanoscale Memory Repair," Springer Publishing Company, Incorporated, 2013.

[22] Y. H. Son, S. Lee, S. O, S. Kwon, N. S. Kim, and J. Ahn, "CiDRA: A Cache-inspired DRAM Resilience Architecture," in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2015

[23] B. Keeth, R. J. Baker, B. Johnson, and F. Lin, "DRAM Circuit Design: Fundamental and High-Speed Topics (2nd ed.)," Wiley-IEEE Press, 2007.

[24] JEDEC, "DDR4 SDRAM Standard," JESD79-4B, 2012.

[25] JEDEC, "288-Pin, 1.2 V (VDD), PC4-1600/PC4-1866/PC4-2133/PC4-2400/PC4-2666/PC4-3200 DDR4 SDRAM Registered DIMM Design Specification," JESD21-C,MODULE4.20.28, 2015.

[26] JEDEC, "288-Pin, 1.2 V (VDD), PC4-1600/PC4-1866/PC4-2133/PC4-2400/PC4-2666/PC4-3200 DDR4 SDRAM Load Reduced DIMM Design Specification," JESD21-C,MODULE4.20.27, 2015.

[27] JEDEC, "DDR4 SDRAM Standard," JESD79-4B, 2012.

[28] JEDEC, "Graphic Double Data Rate 5 (GDDR5) Specification", JESD212C, 2016.

[29] JEDEC, "Low Power Double Data Rate 4 (LPDDR4)," JESD209-4B, 2014.

[30] K. Park, C. Lim, D. Yun, and S. Baeg, "Experiments and Root Cause Analysis for Active-precharge Hammering Fault in DDR3 SDRAM Under 3x nm Technology," Microelectronics

Reliability, vol. 57, 2016.

[31] M. T. Aga, Z. B. Aweke, and T. Austin, “When Good Protections Go Bad: Exploiting Anti-DoS Measures to Accelerate Rowhammer Attacks,” in Proceedings of IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2017.

[32] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, “Flip Feng Shui: Hammering a Needle in the Software Stack,” in USENIX Security Symposium, 2016.

[33] V. van der Veen et al, “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms,” in Proceedings of The ACM Conference on Computer and Communications Security (CCS), 2016.

[34] T. Yang and X-W. Lin, "Trap-Assisted DRAM Row Hammer Effect," in IEEE Electron Device Letters, vol. 40, 2019.

[35] N. Herath and A. Fogh, "These are Not Your Grand Daddys CPU Performance Counters - CPU Hardware Performance Counters for Security," in Black Hat Briefings, 2015.

[36] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," in Black Hat 15, 2015.

[37] H. Kasture and D. Sanchez, "TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications," in Proceedings of IEEE International Symposium on Workload Characterization (IISWC), 2016.

[38] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at

Scale," in Proceedings of 42nd International Symposium on Computer Architecture (ISCA), 2015.

[39] NCSU, "FreePDK45," 2011. [Online]. Available: https://www.eda.ncsu.edu/wiki/FreePDK45:Contents.

[40] Micron, "DDR4 SDRAM System-Power Calculator," 2016.

[41] J. Ahn, S. Li, S. O, and N. P. Jouppi, "McSimA+: A Manycore Simulator with Application-level+ Simulation and Detailed Microarchitecture Modeling," in Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS), 2013.

[42] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist Open-page: A DRAM Page-mode Scheduling Policy for the Many-core Era," in Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2011.

[43] J. L. Henning, "SPEC CPU2006 Memory Footprint," in Computer Architecture News, Vol. 35, 2007.

[44] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2002.

[45] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage," in Proceedings of the 11st USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2014.

[46] S. Beamer, K. Asanovi´c, and D. Patterson, "The GAP Benchmark Suite," arXiv preprint, arXiv:1508.03619, 2015.

[47] PARSEC Group, "A Memo on Exploration of SPLASH-2 Input Sets," Princeton University, 2011.

[48] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," in Proceedings of 35th International Symposium on Computer Architecture (ISCA), 2008.

[49] H. Hassan et al., "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016.

[50] D. Lee et al., "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2015.

[51] W. Shin, J. Yang, J. Choi, and L.-S. Kim, "NUAT: A Non-Uniform Access Time Memory Controller," in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2014.

[52] X. Zhang, Y. Zhang, B. R. Childers, and J. Yang, "Restore Truncation for Performance Improvement in Future DRAM Systems," in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016.

[53] X. Jiang et al., "CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms," in Proceedings of IEEE International Symposium on High Performance Computer

Architecture (HPCA), 2010.

[54] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-stacked and Off-package Memories," in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2015.

[55] Intel, "Intel Xeon Processor 7500 Series Datasheet," 2010.

[56] E. Lee, J. Chung, D. Jung, S. Lee, S. Li, and J. Ahn, "Work as a Team or Individual: Characterizing the System-level Impacts of Main Memory Partitioning," in Proceedings of IEEE International Symposium on Workload Characterization (IISWC), 2017.

[57] C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache," in Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2014.

[58] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent Hardware Management of Stacked DRAM as Part of Memory," in Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2014.

[59] J. Ahn, J. Leverich, R. S. Schreiber, and N. P. Jouppi, "Multicore DIMM: an Energy Efcient Memory Module with Independently Controlled DRAMs," in IEEE Computer Architecture Letters (CAL), vol. 8, 2008.

[60] Y. Ro et al., “SOUP-N-SALAD: Allocation-oblivious Access Latency Reduction with Asymmetric DRAM Microarchitectures,” in Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017.

[61] H. Pilo, V. Ramadurai, G. Braceras, J. Gabric, S. Lamphier, and Y. Tan, “A 450ps Access-time SRAM Macro in 45nm SOI Featuring a Twostage Sensing-scheme and Dynamic Power Management,” in Proceedings of International Solid-State Circuits Conference (ISSCC), 2008.

[62] Synopsys, “Design Compiler: RTL Synthesis,” 2018.

[63] Synopsys, “IC Compiler Place and Route System,” 2018.

[64] JEDEC, “POD12 – 1.2 V PSEUDO OPEN DRAIN INTERFACE,” JESD8-24, 2011.

[65] M. Catrambone, “Routing DDR4 Interfaces Quickly and Efficiently,” in PCB West, 2016.

[66] Micron, “TN-46-14: Hardware Tips for Point-to-Point System Design,” Tech. Rep., 2008.

[67] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2008.

[68] P. Frigo et al., “TRRespass: Exploiting the Many Sides of Target Row Refresh.” In Proceedings of the 41st IEEE Symposium on Security and Privacy (SP), 2020.

국문초록

DRAM을 주기억장치로 사용하는 컴퓨터 시스템은 로우 해머링 공격에 노출된다. 로우 해머링은 인접 DRAM 로우를 자주 activation함으로써 특정 DRAM 로우 데이터에 직접 접근하지 않고서도 데이터를 뒤집을 수 있는 현상을 말한다. 이러한 로우 해머링 현상을 방지하기 위해 여러가지 확률적인 방지 기법과 결정론적 방지 기법들이 연구되어 왔다. 그러나, 확률적인 방지 기법은 공격 자체를 탐지할 수 없고, 방지에 실패할 확률이 0이 아니라는 한계가 있다. 또한 기존의 카운터를 활용한 결정론적 방지 기법들은 큰 칩 면적 비용을 발생시키거나 특정 메모리 접근 패턴에서 현저한 성능 하락을 야기한다는 단점이 있다.

이러한 문제를 해결하기 위해, 우리는 TWiCe (Time Window Counter based row refresh)라는 새로운 카운터 기반 결정론적 방지 기법을 제안한다. TWiCe는 적은 수의 카운터를 활용하여 로우 해머링 공격을 정확하게 탐지하면서도 성능에 악영향을 최소화하는 방법이다. 우리는 DRAM 타이밍 파라미터에 의해 로우 activation 빈도가 제한되고 DRAM 셀이 주기적으로 리프레시 되기 때문에 로우 해머링을 야기할 수 있는 DRAM 로우의 수가 한정된다는 사실에 주목하였다. 이로부터 우리는 TWiCe가 확실한 결정론적 방지를 보장할 경우 필요한 DRAM 뱅크 당 필요한 카운터 수의 최대값을 구하였다. TWiCe는 일반적인 DRAM 동작 과정에서는 성능에 아무런 영향을 미치지 않으며, 현대 DRAM 디바이스에서 0.7% 이하의 칩 면적 증가 및 에너지 증가만을 필요로 한다. 우리가 진행한 평가에서 TWiCe는 로우 해머링

공격 시나리오를 포함한 여러가지 메모리 접근 패턴에서 0.006% 이하의 추가적인 DRAM activation을 요구하였다.

또한 TWiCe의 칩 면적 및 에너지 비용을 더욱 줄이기 위하여, 우리는 threshold가 조정된 랭크 단위 TWiCe를 제안한다. 먼저, 수백개가 넘는 TWiCe 테이블 항목 검색을 에너지 효율적으로 수행할 수 있는 pa-TWiCe (pseudo-associatvie TWiCe)를 제안하였다. 그리고, 테이블 항목을 랭크 단위로 관리하여 필요한 테이블 항목의 수를 더욱 줄인 랭크 단위 TWiCe를 제안하였다. 또한, 우리는 TWiCe의 threshold 값을 조절함으로써 일반적인 워크로드 상에서 거짓 양성(false-positive) 탐지를 증가시키지 않는 선에서 TWiCe의 테이블 항목 수를 더욱 줄였다.

마지막으로, 우리는 컴퓨터 시스템의 주기억장치 성능 향상을 위해 TWiCe를 hot-page 감지기로 사용하는 것을 제안한다. 메모리 접근의 시간적 지역성에 의해 최근 자주 activation된 DRAM 로우들은 다시 activation될 확률이 높고, TWiCe는 최근 자주 activation된 DRAM 로우에 대한 정보를 가지고 있다. 이러한 사실에 기반하여, 우리는 hot-page에 대한 DRAM 접근 지연시간을 줄이는 DRAM 페이지 스왑(swap) 기법들에 TWiCe를 적용하는 방법을 보인다. 우리가 수행한 평가에서 TWiCe를 사용한 저지연시간 DRAM은 멀티 쓰레딩 워크로드들에서 기존 DDR4 디바이스 대비 IPC를 최대 12.2% 증가시켰다.

주요어 : DRAM, 로우 해머링, 결정론적 방지, 신뢰성, 핫-페이지 감지, 저지연시간 DRAM
학 번 : 2014-24902